Compilation of Julia code for deployment in Model-Based Engineering workflows.

Fredrik Bagge Carlson, MODPROD February, 2025

Outline

- What is Julia?
- The Julia compilation pipeline
- Ahead-of-time compilation of Julia
  - Historically
  - Now and near future
- Demos
  - Executable (state estimation)
  - Shared library (PID-controller library)
- Current limitations

Julia Language

julia: A first look?

```julia
julia> struct Circle
           r::Float64
       end

julia> area(c::Circle) = c.r^2 * π;

julia> struct Rectangle
           width::Float64
           height::Float64
       end

julia> area(r::Rectangle) = r.width * r.height;

julia> shapes = [Rectangle(2, 3),
                 Circle(2),
                 Rectangle(5, 2),];

julia> area.(shapes)
3-element Vector{Float64}:
  6.0
 12.566370614359172
 10.0
```

```julia
julia> f(x) = x^2 + x;

julia> f(2)
6

julia> f(3.0)
12.0

julia> f(2 + im)
5 + 5im

julia> f([1 2; 2 3])
2×2 Matrix{Int64}:
  6  10
 10  16

julia> using ForwardDiff: Dual

julia> f(Dual(3, 1))
Dual(12,7)
```

julia: Specialization by compiler

```
julia> @code_native f(2)
        imulq   %rdi, %rax
        addq    %rdi, %rax
...

julia> @code_native f(3.0)
        vmulsd  %xmm0, %xmm0, %xmm1
        vaddsd  %xmm0, %xmm1, %xmm0
...

julia> @code_native f(2 + im)
...
        imulq   %rcx, %rsi
        movq    %rdx, %rdi
        imulq   %rcx, %rdi
        leaq    (%rdx,%rdi,2), %rdi
        imulq   %rdx, %rdx
        addq    %rcx, %rsi
        subq    %rdx, %rsi
...

julia> @code_native f(Dual(2,3))
...
        imulq   %rcx, %rsi
        addq    %rcx, %rsi
        imulq   %rdx, %rcx
...
```

Julia compiler pipeline

Julia normally compiles “just ahead of time”


```julia
function foo(a)
    b = a > 0 ? 1 : randn(2,2)
    sin(b)
end
```

```
julia> foo(-1)
2×2 Matrix{Float64}:
 0.0312928  0.60576
 1.2603     0.201017

julia> foo(11)
0.8414709848078965
```

Ahead-of-time (AOT) compilation and distribution of Julia programs

Historically, either of:

- Distribute the **source code**
- Package ***everything*** into a ***huge*** binary
  - Package and application source
  - Compiled code
  - Julia compiler
  - LLVM compiler
  - Julia runtime

Benefits:

👍 *All* language features are intact

👍 Self contained

Drawbacks:

👎 Source distribution requires Julia install

👎 Not guaranteed to be AOT compiled

👎 The generated artifact is huge (GB)

👎 Not (always) relocatable

Ahead-of-time (AOT) compilation and distribution of Julia programs

Now and near future:

- Remove everything that isn't reachable from entry point (trimming)
- Complain if uncompilable
  - Eval
  - Types unknown

Benefits:

👍 *May* produce smallish binaries (~900KB hello world)
👍 Guarantees AOT compilation
👍 Most language features intact (no eval, no unbounded dispatch)

Current drawbacks:

👎 Not yet released
👎 No easy cross-compilation
👎 Not yet self contained (link to `libjulia`)

Ahead-of-time (AOT) compilation and distribution of Julia programs

Would this be okay?

```julia
function foo(a)
    b = a > 0 ? 1 : randn(2,2)
    sin(b)
end
```

```
julia> foo(-1)
2×2 Matrix{Float64}:
 0.0312928  0.60576
 1.2603     0.201017

julia> foo(11)
0.8414709848078965
```

```
julia> @code_warntype foo(1)
MethodInstance for foo(::Int64)
  from foo(a) @ Main REPL[38]:1
Arguments
  #self#::Core.Const(Main.foo)
  a::Int64
Locals
  b::Union{Int64, Matrix{Float64}}
  @_4::Union{Int64, Matrix{Float64}}
Body::Union{Float64, Matrix{Float64}}
1 ─       Core.NewvarNode(:(b))
│   %2  = Main.:>::Core.Const(>)
│   %3  = (%2)(a, 0)::Bool
└──       goto #3 if not %3
2 ─       (@_4 = 1)
└──       goto #4
3 ─ %7  = Main.randn::Core.Const(randn)
└──       (@_4 = (%7)(2, 2))
4 ┄ %9  = @_4::Union{Int64, Matrix{Float64}}
│         (b = %9)
│   %11 = b::Union{Int64, Matrix{Float64}}
│   %12 = Main.sin(%11)::Union{Float64, Matrix{Float64}}
└──       return %12
```

Demos

- Executable (State estimation)
  - Equation-based model
  - State estimator
  - Executable that loads a data file from disk and performs filtering
- Shared library (PID controller)
  - Julia PID controller library
  - Expose library functions with C-compatible interface
  - Compile shared library
  - Load library from C program and call functions

Model-based state estimation

Demonstrate use of

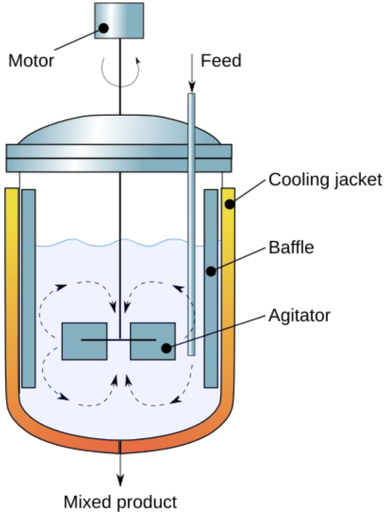
Equation-based modeling with ModelingToolkit
(Could be a model from OpenModelica)

+

Off-the-shelf Julia library for state estimation

↓

Executable



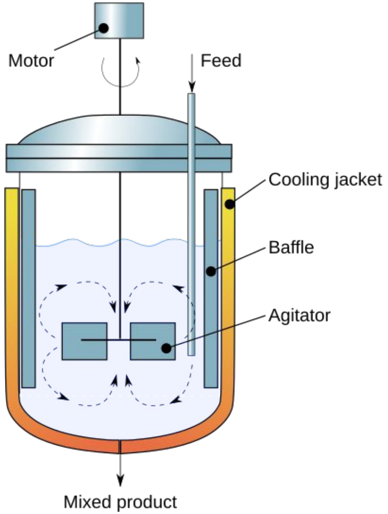
Source available: https://github.com/baggepinnen/static_kalman

Model-based state estimation

```julia
@mtkmodel CSTR begin
    @variables begin
        Cₐ(t), [description = "Concentration of reactant A"],
        Cᵦ(t), [description = "Concentration of reactant B"],
        Tᵣ(t), [description = "Temperature in reactor"],
        Tₖ(t), [description = "Temperature in cooling jacket"],
        F(t),  [description = "Feed", input=true],
        Q̇(t),  [description = "Heat flow", input=true]
    end
    begin
        K₁ = K0_ab * exp((-E_A_ab)/((Tᵣ+273.15)))
        K₂ = K0_bc * exp((-E_A_bc)/((Tᵣ+273.15)))
        K₃ = K0_ad * exp((-E_A_ad)/((Tᵣ+273.15)))
        TΔ = Tᵣ-Tₖ
    end
    @equations begin
        D(Cₐ) ~ F*(C_A0 - Cₐ)-K₁*Cₐ - K₃*abs2(Cₐ)
        D(Cᵦ) ~ -F*Cᵦ + K₁*Cₐ - K₂*Cᵦ
        D(Tᵣ) ~ ((K₁*Cₐ*Hᵣ_ab + K₂*Cᵦ*Hᵣ_bc + K₃*abs2(Cₐ)*Hᵣ_ad)/(-Rou*Cp)) +
                  F*(T_in-Tᵣ) + (((K_w*Aᵣ)*(-TΔ))/(Rou*Cp*Vᵣ))
        D(Tₖ) ~ (Q̇ + K_w*Aᵣ*(TΔ))/(m_k*Cpₖ)
    end
end
```

Generate Julia code

```julia
@named model = CSTR()
cmodel = complete(model)
inputs = [cmodel.F, cmodel.Q̇]
(f_oop, f_ip), x_sym, p, io_sys = ModelingToolkit.generate_control_function(model, inputs)
```



Source available: https://github.com/baggepinnen/static_kalman

Model-based state estimation

```julia
module StateEstimator

using LowLevelParticleFilters
using Random, LinearAlgebra, StaticArrays

const Ts  = 0.005 # sample time
const x0  = SA[0.8, 0.5, 134.14, 130] # Initial state
const u0 = SA[12.0, -4000] # Initial input
const p = nothing

measurement(x,u,p,t) = x # We can measure the full state
include("dynamics.jl") # This defines variable "dynamics"
const discrete_dynamics = LowLevelParticleFilters.rk4(dynamics, Ts)

const nx = 4 # Dimension of state
const nu = 2 # Dimension of input
const ny = 4 # Dimension of measurements

const R1 =   SA[5.79056e-5  -6.10652e-6  -0.00449048  -0.00129742
               -6.10652e-6   1.1864e-5    0.00115109   0.0003243
               -0.00449048   0.00115109   0.770544     0.185088
               -0.00129742   0.0003243    0.185088     0.612284]
const R2 = SMatrix{nx,nx}(Diagonal(x0 .^ 2 ./ 10))
const d0 = LowLevelParticleFilters.SimpleMvNormal(x0,R1)   # Initial state Distribution

const kf = UnscentedKalmanFilter(discrete_dynamics, measurement, R1, R2, d0; ny, nu, p)

Base.@ccallable function main()::Cint
    println(Core.stdout, "I'm alive and well")

    y = reinterpret(SVector{4, Float64}, read("data_y.bin"))
    u = reinterpret(SVector{2, Float64}, read("data_u.bin"))
    @assert length(y) == length(u)
    println(Core.stdout, "I read the data, it has length ", length(y))

    sol = forward_trajectory(kf, u, y)
    println(Core.stdout, "I got loglik = ", sol.ll)

    return zero(Cint)
end
end
```

Include generated Julia code

Discretize

Define state estimator

Binary size: 3.5MB

Runtime: 27ms

Of which filtering is 62μs

```
julia> run(`./juliac_demo`)
I'm alive and well
I read the data, it has length 30
I got loglik = -238.82851486636454
```

Julia package as shared library

Demonstrate use of

Off-the-shelf julia package for PID controllers

↓

Executable

↓

Loaded and called from C program

Source available: https://github.com/JuliaControl/DiscretePIDs.jl

Julia package as shared library

Expose library functions as C-callable (entrypoints)

```julia
# Set the initial PID parameters here
const pid = DiscretePIDs.DiscretePID(; K = T(1), Ti = 1, Td = false, Ts = 1)


@ccallable function calculate_control!(r::T, y::T, uff::T)::T
    DiscretePIDs.calculate_control!(pid, r, y, uff)::T
end

@ccallable function set_K!(K::T, r::T, y::T)::Cvoid
    DiscretePIDs.set_K!(pid, K, r, y)
    nothing
end

@ccallable function set_Ti!(Ti::T)::Cvoid
    DiscretePIDs.set_Ti!(pid, Ti)
    nothing
end

@ccallable function set_Td!(Td::T)::Cvoid
    DiscretePIDs.set_Td!(pid, Td)
    nothing
end

@ccallable function reset_state!()::Cvoid
    DiscretePIDs.reset_state!(pid)
    nothing
end
```

Shared-object file size:
1.7MB

Source available: https://github.com/JuliaControl/DiscretePIDs.jl

Julia package as shared library

Load compiled library and call from C

```c
#define LIB_PATH "/home/fredrikb/.julia/dev/DiscretePIDs/examples/juliac/juliac_pid.so"
void *lib_handle = dlopen(LIB_PATH, RTLD_LAZY);
jl_init_with_image_t jl_init_with_image = (jl_init_with_image_t)dlsym(lib_handle, "jl_init_with_image");
jl_init_with_image(JULIA_PATH, LIB_PATH);
```

For loading compiled library and Julia runtime (not yet fully self contained)

```c
// Trivial test program that computes a few control outputs and modifies K
double r = 1.0, y = 0.0, uff = 0.0;
double result = calculate_control(r, y, uff);
printf("calculate_control! returned: %f\n", result);
result = calculate_control(r, y, uff);
printf("calculate_control! returned: %f\n", result);
set_K(0.0, r, y);
for (int i = 0; i < 3; ++i) {
    result = calculate_control(r, y, uff);
    printf("calculate_control! returned: %f\n", result);
}
```

Compile and link to libjulia

```
gcc -o pid_program test_juliac_pid.c -I …/julia/usr/include/julia
-L…/julia/usr/lib -ljulia -ldl
```

Source available: https://github.com/JuliaControl/DiscretePIDs.jl

Deployment on a Raspberry Pi

- The same workflows can be performed *on* a Raspberry Pi (or similar device)
- Binary runtime (state estimation) about 4x slower on RPi
- Currently no first-class support for cross compilation
- Compilation in an emulator is a viable option in some cases

Current limitations

- Julia runtime still required → only works on supported platforms
  - ✅ Traditional OS (Linux, Win, MacOS)
  - ✅ x86-64
  - ✅ ARMv8
  - 🟡 ARMv7
  - 🟡 RISC-V
  - 🟥 Real-time OSes
  - 🧙 Arduino?
- Runtime is not yet trimmed
- Not released
- Cross compilation
- All of these limitations are being worked on

Should you use this *today*?

Are you a Julia hacker?

✅ Maybe

❌ Hold off until release

Summary

- Julia can now be ahead-of-time compiled to a small binary
- Most features of Julia are intact while doing so
- Restrictions around too much dynamism
- Not yet released in stable julia version (v1.12 feature freeze was a month ago)
- Impact on size of Julia FMUs