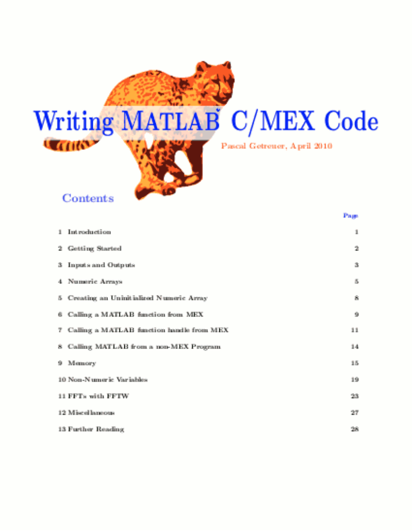
A fresh approach to Numerical Computing

Zachary Sunberg, Assistant Professor, AERO

Survey

Scientific computing is pretty good these days

Fast Development | Fast Execution



Writing MATLAB C/MEX Code

Pascal Getreuer, April 2010

Contents

2011

Zachary Sunberg
@zsunberg

I think #python might be the programing language I've always wanted.

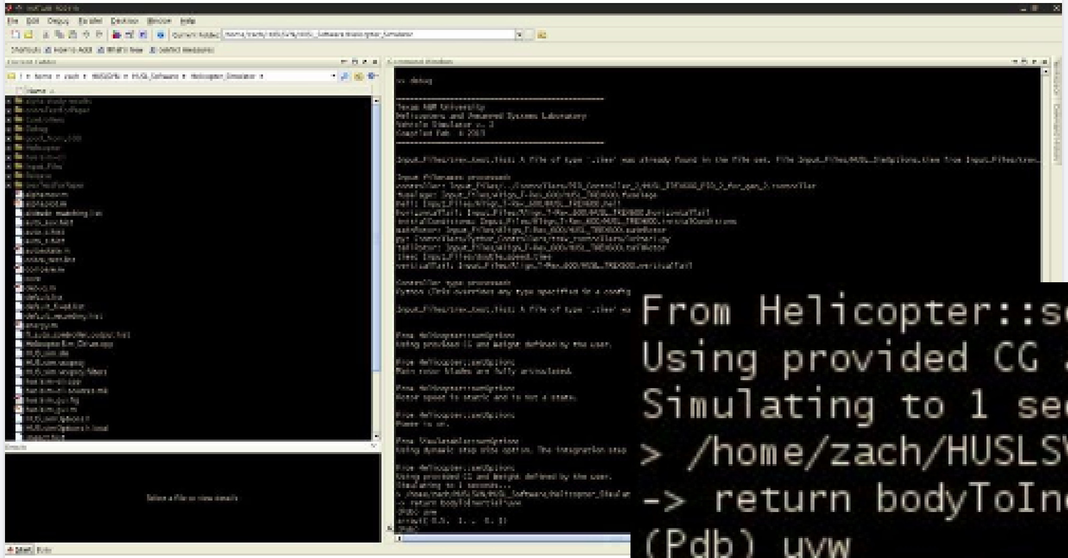
1:25 AM - Feb 3, 2011

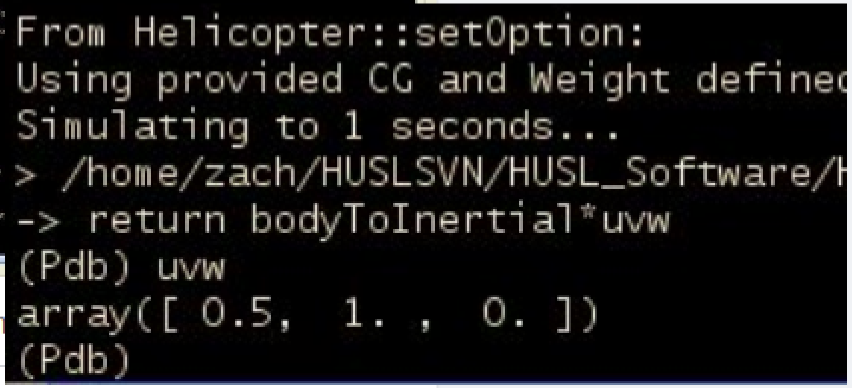
See Zachary Sunberg's other Tweets

2013

Zach Sunberg
February 5, 2013 ·

Codeception: Interactively debugging a python code that's running inside an application written in C++ that I'm running in the matlab command prompt.





You, Micki Achterhof, David Freese an

Like Comment Share

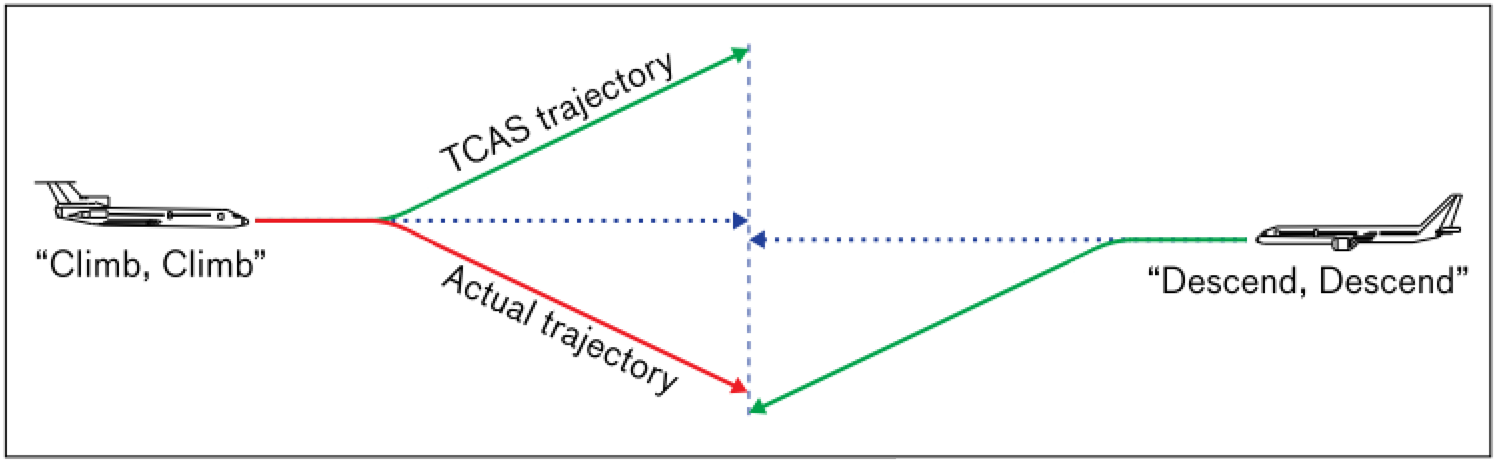
Write a comment...

2012

We love [Matlab, Lisp, Python, Ruby, Perl, Mathematica, and C]; they are wonderful and powerful. For the work we do — scientific computing, machine learning, data mining, large-scale linear algebra, distributed and parallel computing — each one is perfect for some aspects of the work and terrible for others. **Each one is a trade-off.**

We are greedy: we want more.

Also ~2011: Improving TCAS




```
IF (ITF.A LT G.ZTHR)
  THEN IF(ABS
  (ITF.VMD) LT
  G.ZTHR)
    THEN SET ZHIT;
    ELSE CLEAR
```

Algorithm 11 ProximityEstimation

```
function ProximityEstimation( z_own_ave::R, z_int_ave::R, r_ground_int::R, valid_bearing_int::Bool,
                      belief_vert_int::Vector{IntruderVerticalBelief(p. 102)}, quant_int::Z )
    D_proximity_range_threshold::R = params().display.proximity_range_threshold
    H_proximity_altitude_threshold = params().display.proximity_altitude_threshold
    is_proximate::Bool = false
    z_rel::R = abs( z_own_ave - z_int_ave )
    if (r_ground_int <= D_proximity_range_threshold) && (z_rel < H_proximity_altitude_threshold)
        if !IsNARS(p. 95)( belief_vert_int, quant_int ) || valid_bearing_int
            is_proximate = true
        end
    end
    return is_proximate::Bool
end
```

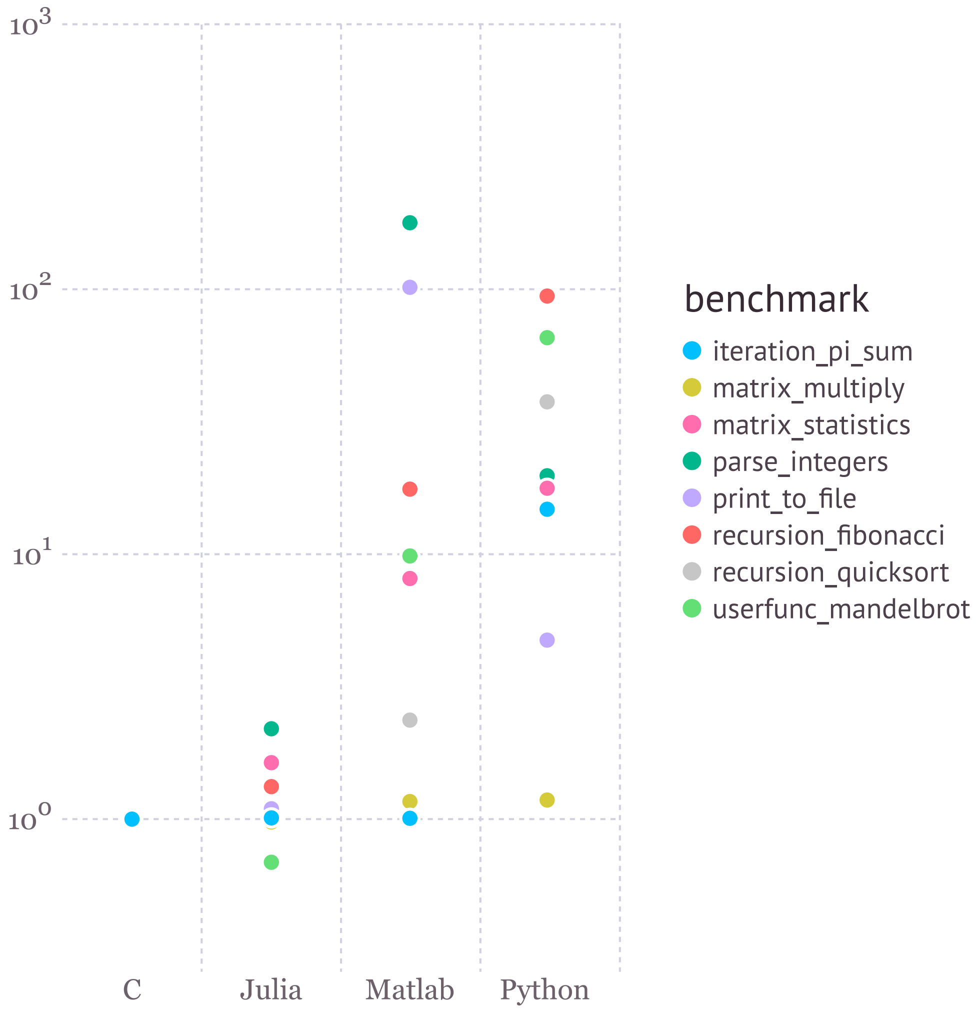
Julia solves the two language problem!

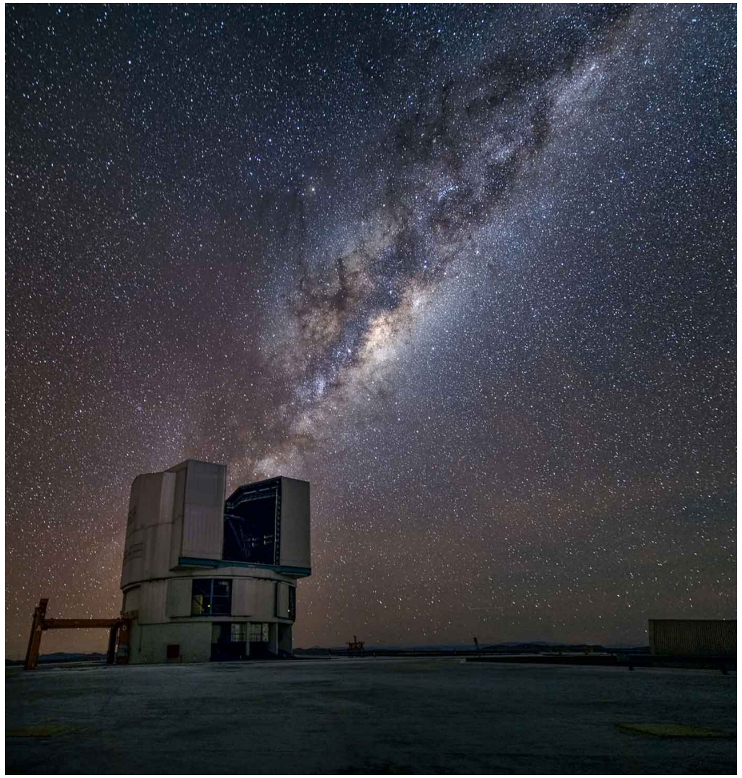
- ~~as easy~~ **EASIER** to read, write, maintain, and collaborate with as Python or MATLAB
- ~~as fast~~ **FASTER** as C

Be skeptical! Find out for yourself!

Others' perspective: https://web.mit.edu/18.06/www/Spring17/Julia-intro.pdf

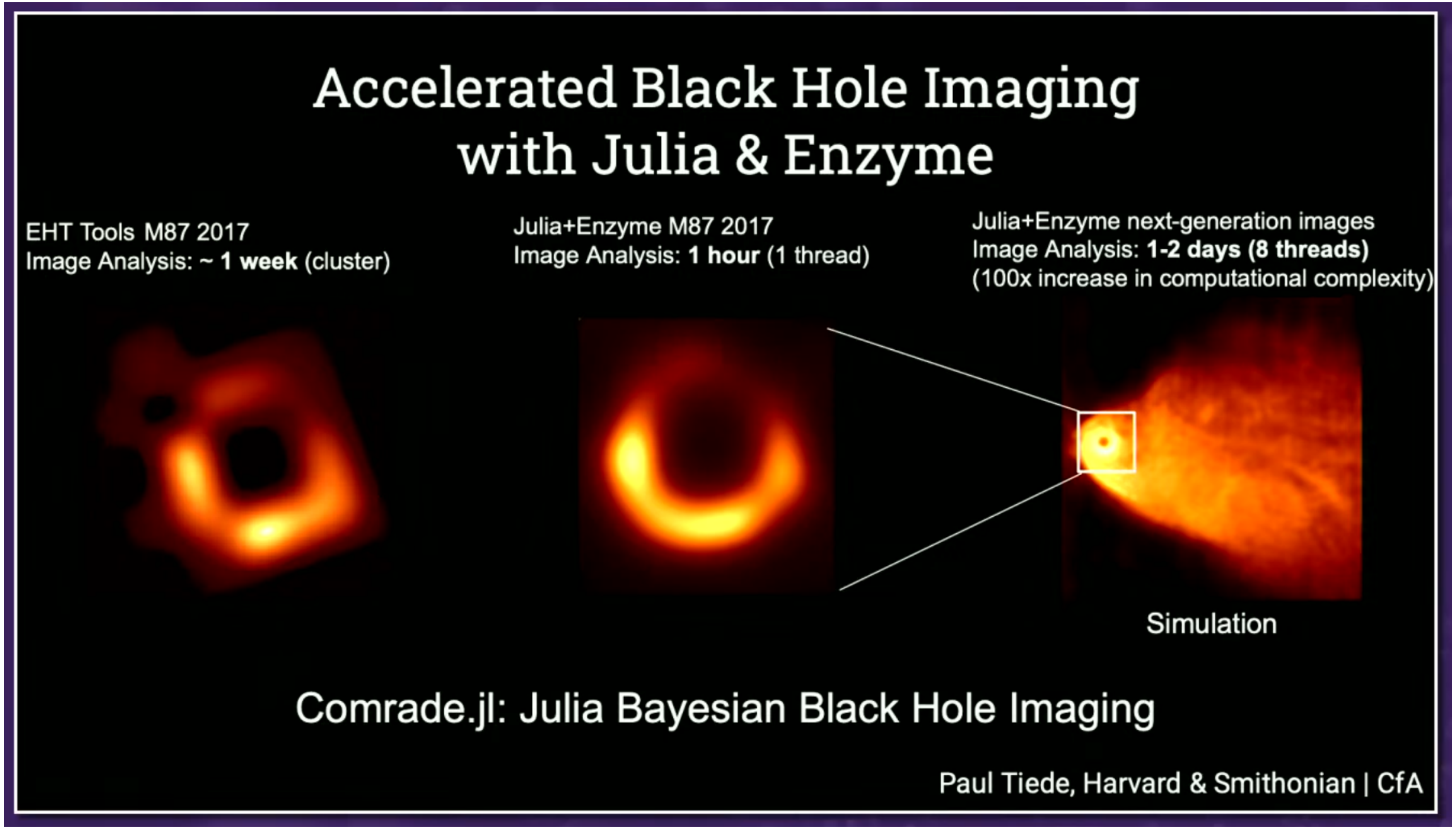
Julia - Speed





Celeste Project

1.54 Petaflops



JuliaCon 2023 (EHT Tools in C++)

Not because Julia is somehow faster, but because it was easier to dev better algs

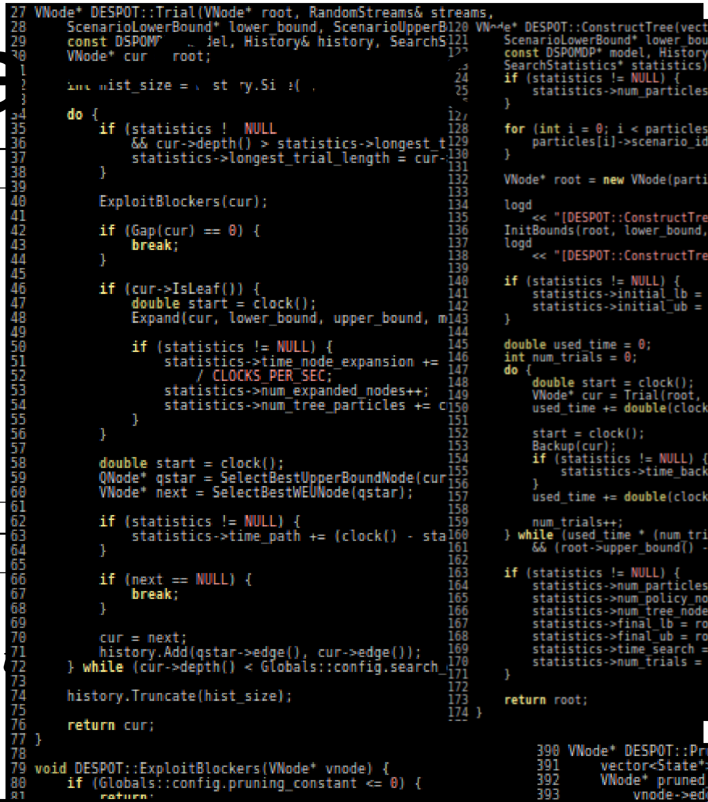
Julia

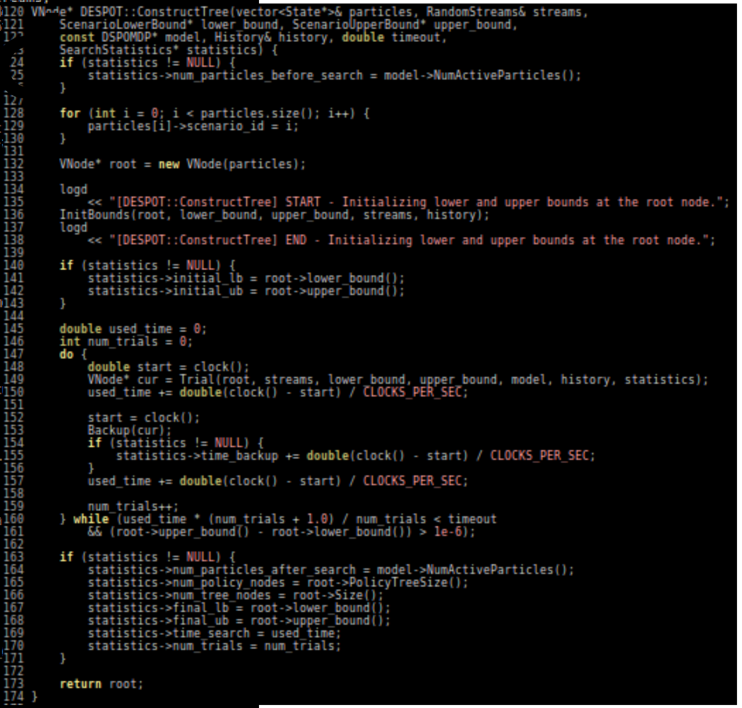
Algorithm 1 BUILDDESPOT(b_0)

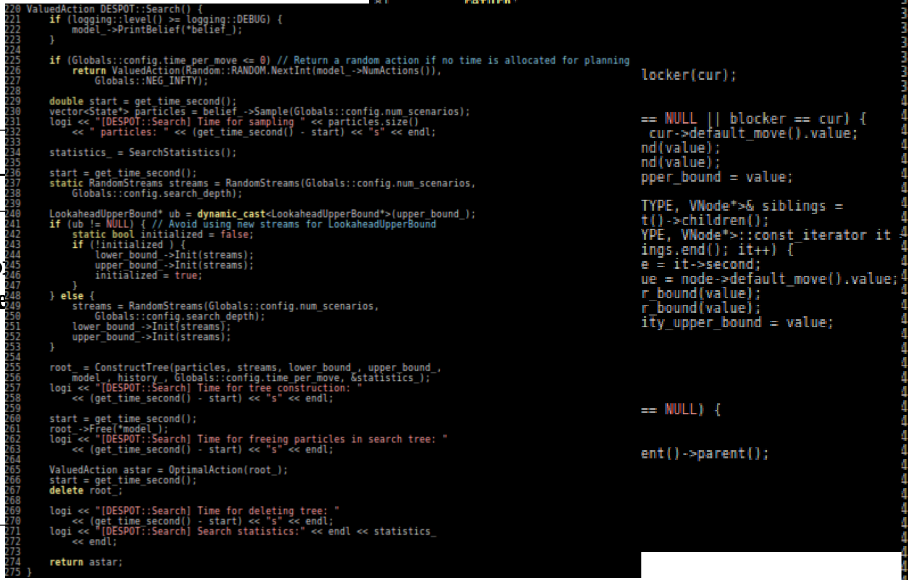
1: Sample randomly a set Φ_{b_0} of K scenarios from the current belief b_0.
2: Create a new DESPOT $\mathcal{D}$ with a single node b_0 as the root.
3: Initialize $U(b_0), L_0(b_0), \mu(b_0),$ and $\ell(b_0)$.
4: $\epsilon(b_0) \leftarrow \mu(b_0) - \ell(b_0)$.
5: **while** $\epsilon(b_0) > \epsilon_0$ and the total running time is less than $T_{\max}$ **do**
6: $\quad b \leftarrow$ EXPLORE($\mathcal{D}, b_0$).
7: $\quad$ BACKUP($\mathcal{D}, b$).
8: $\quad \epsilon(b_0) \leftarrow \mu(b_0) - \ell(b_0)$.
9: **return** ℓ

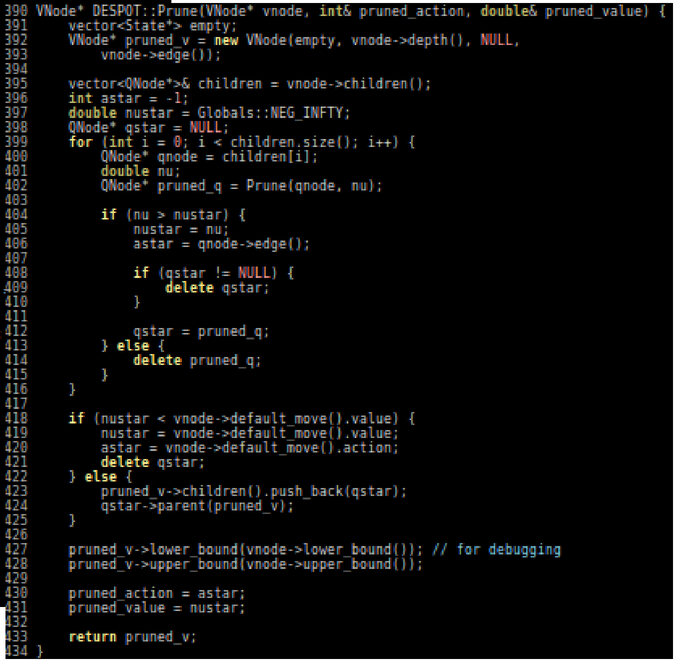
Algorithm 2 EXPLORE($\mathcal{D}, b$)

1: **while** $\Delta(b) \leq D$, $E(b) > 0$, and PRUNE($\mathcal{D}, b$) = FALSE **do**
2: $\quad$ **if** b is a leaf node in $\mathcal{D}$ **then**
3: $\quad\quad$ Expand b one level deeper. Insert each new child b' of b into $\mathcal{D}$. Initialize
$\quad\quad \mu(b')$, and $\ell(b')$.
4: $\quad a^* \leftarrow \arg\max_{a \in A} \mu(b, a)$.
5: $\quad z^* \leftarrow \arg\max_{z \in Z_{b,a^*}} E(\tau(b, a^*, z))$.
6: $\quad b \leftarrow \tau(b, a^*, z^*)$.
7: **if** $\Delta(b) > D$ **then**
8: $\quad$ MAKEDEFAULT(b).
9: **return** b.









Algorithm 4 PRUNE($\mathcal{D}, b$)

1: BLOCKED $\leftarrow$ FALSE.
2: **for** each node x on the path from b to the root o
3: $\quad$ **if** b' is blocked by any ancestor node in $\mathcal{D}$ **the**
4: $\quad\quad$ MAKEDEFAULT(x).
5: $\quad\quad$ BACKUP($\mathcal{D}, x$).
6: $\quad\quad$ BLOCKED $\leftarrow$ TRUE.
7: $\quad$ **else**
8: $\quad\quad$ **break**
9: **return** BLOCKED

Julia - S

Algorithm 1 BUILDDESPOT(b_0)

1: Sample randomly a set Φ_{b_0} of K scenarios from the current belief b_0.
2: Create a new DESPOT $\mathcal{D}$ with a single node b_0 as the root.
3: Initialize $U(b_0), L_0(b_0), \mu(b_0),$ and $\ell(b_0)$.
4: $\epsilon(b_0) \leftarrow \mu(b_0) - \ell(b_0)$.
5: **while** $\epsilon(b_0) > \epsilon_0$ and the total running time is less than $T_{\max}$ **do**
6: $\quad b \leftarrow$ EXPLORE($\mathcal{D}, b_0$).
7: $\quad$ BACKUP($\mathcal{D}, b$).
8: $\quad \epsilon(b_0) \leftarrow \mu(b_0) - \ell(b_0)$.
9: **return** ℓ

Algorithm 2 EXPLORE($\mathcal{D}, b$)

1: **while** $\Delta(b) \leq D$, $E(b) > 0$, and PRUNE($\mathcal{D}, b$) = FALSE **do**
2: $\quad$ **if** b is a leaf node in $\mathcal{D}$ **then**
3: $\quad\quad$ Expand b one level deeper. Insert each new child b' of b into $\mathcal{D}$. Initialize $U(b')$, $L_0(b')$, $\mu(b')$, and $\ell(b')$.
4: $\quad a^* \leftarrow \arg\max_{a \in A} \mu(b, a)$.
5: $\quad z^* \leftarrow \arg\max_{z \in Z_{b,a^*}} E(\tau(b, a^*, z))$.
6: $\quad b \leftarrow \tau(b, a^*, z^*)$.
7: **if** $\Delta(b) > D$ **then**
8: $\quad$ MAKEDEFAULT(b).
9: **return** b.

Algorithm 4 PRUNE($\mathcal{D}, b$)

1: BLOCKED $\leftarrow$ FALSE.
2: **for** each node x on the path from b to the root of $\mathcal{D}$ **do**
3: $\quad$ **if** b' is blocked by any ancestor node in $\mathcal{D}$ **then**
4: $\quad\quad$ MAKEDEFAULT(x).
5: $\quad\quad$ BACKUP($\mathcal{D}, x$).
6: $\quad\quad$ BLOCKED $\leftarrow$ TRUE.
7: $\quad$ **else**
8: $\quad\quad$ **break**
9: **return** BLOCKED

```julia
function build_despot(p::DESPOTPlanner, b_0)
    D = DESPOT(p, b_0)
    b = 1
    trial = [illegible]
    start = CPUtime_us()

    while D.mu[1]-D.l[1] > p.sol.epsilon_0 &&
          CPUtime_us()-start < p.sol.T_max*1e6 &&
          trial <= p.sol.max_trials
        b = explore!(D, 1, p)
        backup!(D, b, p)
        trial += 1
    end

    return D
end

function explore!(D::DESPOT, b::Int, p::DESPOTPlanner)
    while D.Delta[b] <= p.sol.D &&
        excess_uncertainty(D, b, p) > 0.0
        && !prune!(D, b, p)

        if isempty(D.children[b]) # a leaf
            expand!(D, b, p)
        end
        b = next_best(D, b, p)
    end
    if D.Delta[b] > p.sol.D
        make_default!(D, b)
    end
    return b
end

function prune!(D::DESPOT, b::Int, p::DESPOTPlanner)
    x = b
    blocked = false
    while x != 1
        n = find_blocker(D, x, p)
        if n > 0
            make_default!(D, x)
            backup!(D, x, p)
            blocked = true
        else
            break
        end
        x = D.parent_b[x]
    end
    return blocked
end
```

How does it work?

Speed: Just-in-time LLVM Compilation