Getting Julia Ready for Statistical Computing

John Myles White

April 27, 2012

Julia is a promising new language

At present, its strength is linear algebra, not data analysis

What do we need to start doing statistical computing in Julia?

Some examples:

- NA type (Harlan Harris)

Some examples:

- NA type (Harlan Harris)
- DataFrame's (Harlan Harris)

Some examples:

- NA type (Harlan Harris)
- DataFrame's (Harlan Harris)
- Statistical distribution functions (Doug Bates)

Some examples:

- NA type (Harlan Harris)
- DataFrame's (Harlan Harris)
- Statistical distribution functions (Doug Bates)
- GLM's (Doug Bates)

Some examples:

- NA type (Harlan Harris)
- DataFrame's (Harlan Harris)
- Statistical distribution functions (Doug Bates)
- GLM's (Doug Bates)
- Hypothesis testing (JMW)

Some examples:

- NA type (Harlan Harris)
- DataFrame's (Harlan Harris)
- Statistical distribution functions (Doug Bates)
- GLM's (Doug Bates)
- Hypothesis testing (JMW)
- Optimization (JMW)

Let's talk about implementing optimization algorithms in Julia

Simulated Annealing is a randomized search method

SA is computationally intensive and a perfect candidate for Julia

```
function simulated_annealing(cost::Function,
                             s0::Any,
                             neighbor::Function,
                             temperature::Function,
                             iterations::Int64,
                             keep_best::Bool)

  # Set our current state to the specified intial state.
  s = s0

  # Set the best state we've seen to the intial state.
  best_s = s0
```

```
# We always perform a fixed number of iterations.
for i = 1:iterations

  # Find the proper temperature at time i.
  t = temperature(i)

  # Randomly generate a neighbor of our current state.
  s_n = neighbor(s)

  # Evaluate the cost function.
  y = cost(s)
  y_n = cost(s_n)
```

```
if y_n <= y
  # We always move to superior states.
  s = s_n
else
  # We probabilistically move to inferior states.
  p = exp(- ((y_n - y) / t))

  if rand() <= p
    s = s_n
  else
    s = s
  end
end
```

```
    # Keep a record of the best state we've seen.
    if cost(s) < cost(best_s)
      best_s = s
    end
  end

  # Return the best state or the last state we've seen.
  if keep_best
    best_s
  else
    s
  end
end
```

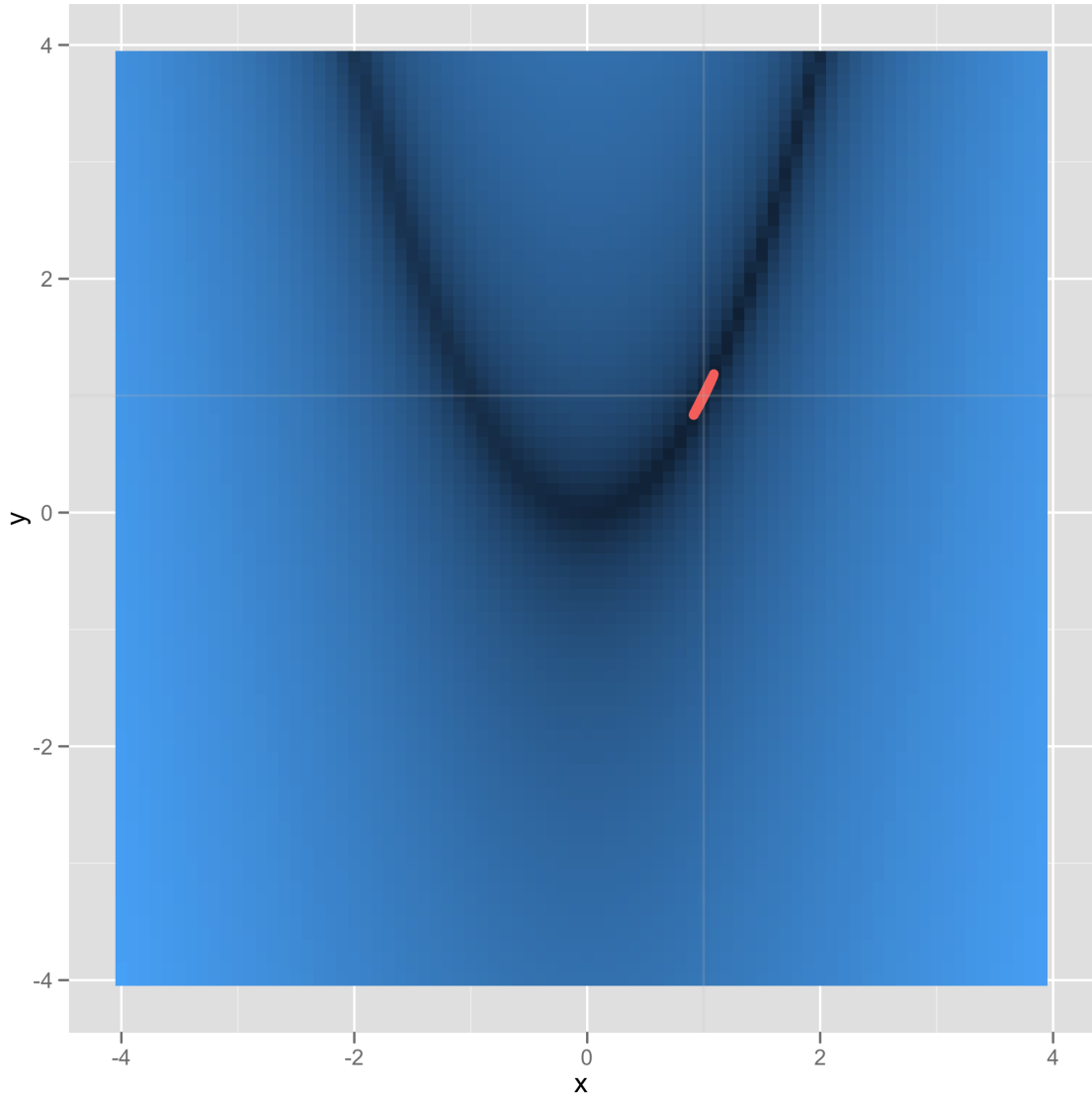
That very generic piece of code runs quite fast in Julia

```
function rosenbrock(x, y)
  (1 - x)^2 + 100(y - x^2)^2
end

function neighbors(z)
  [rand_uniform(z[1] - 1, z[1] + 1),
   rand_uniform(z[2] - 1, z[2] + 1)]
end

simulated_annealing(z -> rosenbrock(z[1], z[2]),
                    [0, 0],
                    neighbors,
                    i -> 1 / log(i),
                    10000,
                    true)
```

We can run this thousands of times to see how well it performs


4
2
0
-2
-4
y
-4
-2
0
2
4
x

We can also estimate average run-time of 10,000 iterations

```
@elapsed for i = 1:5000
  simulated_annealing(z -> rosenbrock(z[1], z[2]),
                      [0, 0],
                      neighbors,
                      i -> 1 / log(i),
                      10000,
                      true)
end
```

Each run of 10,000 iterations takes 25 milliseconds in Julia

A line-by-line translation into R takes 500 milliseconds per run

Julia has a lot of promise, but it needs you

Julia needs people to implement algorithms

Julia needs feedback on language design