



123 lines - 9 Removals

```

1 
2 """
3     helpers.jl -- Feel free to modify anything you
      like in this file!
4 
5 Provides the problem structure and some useful fun
      ctions. In particular, students may be
6 interested in `count` - to help with writing their
      algorithm, and `main` - to test their
7 methods. Note that `starter_code/localtest.jl` pro
      vides an identical test to that found
8 in the autograder. `main`, on the other hand, retu
      rns other metrics that may be of
9 interest during development.
10 """
11 
12 # Statistics is the only allowed external package
      (Random is part of the standard library)
13 # If you would like to use any other (standard) pa
      ckages, make sure to do so in `project2.jl`
14 # *NOT HERE*.
15 using Random
16 using Statistics
17 
18 # A global counter that keeps track of how many ti
      mes each function has been called.
19 # It may seem like a clever hack to edit this dict
      ionary as part of your optimize
20 # method to get infinite function evaluations, but
      beware...
21 # you'll regret it when it goes through the autogr
      ader
22 const COUNTERS = Dict{String, Int}{}
23 
24 """
25     @counted
26 
27 A function defined with this macro increments the
      global counter `COUNTERS`
28 each time it's called.
29 
30 Example:
31     @counted f(x) = 2x #each time `f(x)` is calle
      d, we also have `COUNTERS["f"] += 1`
32 """
33 macro counted(f)
34     name = f.args[1].args[1]
35     name_str = String(name)
36     body = f.args[2]
37     update_counter = quote
38         if !haskey(COUNTERS, $name_str)
39             COUNTERS[$name_str] = 0
40         end
41         COUNTERS[$name_str] += 1
42     end
43     insert!(body.args, 1, update_counter)
44     return f

```

114 lines 0 Additions

```

1 """
2     helpers.jl -- Feel free to modify anything you
      like in this file!
3 
4 Provides the problem structure and some useful fun
      ctions. In particular, students may be
5 interested in `count` - to help with writing their
      algorithm, and `main` - to test their
6 methods. Note that `starter_code/localtest.jl` pro
      vides an identical test to that found
7 in the autograder. `main`, on the other hand, retu
      rns other metrics that may be of
8 interest during development.
9 """
10 
11 # Statistics is the only allowed external package
      (Random is part of the standard library)
12 # If you would like to use any other (standard) pa
      ckages, make sure to do so in `project2.jl`
13 # *NOT HERE*.
14 using Random
15 using Statistics
16 
17 # A global counter that keeps track of how many ti
      mes each function has been called.
18 # It may seem like a clever hack to edit this dict
      ionary as part of your optimize
19 # method to get infinite function evaluations, but
      beware...
20 # you'll regret it when it goes through the autogr
      ader
21 const COUNTERS = Dict{String, Int}{}
22 """
23     @counted
24 
25 A function defined with this macro increments the
      global counter `COUNTERS`
26 each time it's called.
27 
28 Example:
29     @counted f(x) = 2x #each time `f(x)` is calle
      d, we also have `COUNTERS["f"] += 1`
30 """
31 macro counted(f)
32     name = f.args[1].args[1]
33     name_str = String(name)
34     body = f.args[2]
35     update_counter = quote
36         if !haskey(COUNTERS, $name_str)
37             COUNTERS[$name_str] = 0
38         end
39         COUNTERS[$name_str] += 1
40     end
41     insert!(body.args, 1, update_counter)
42     return f

```

```

45 end
46
47
48 # simple.jl defines the 3 simple problems. It's in
  cluded down here rather than at the
49 # top because it relies on @counted and COUNTERS t
  o track the evaluation counts.
50 include("simple.jl")
51
52 const PROBS = Dict{"simple1" => (f=simple1, g=simp
  le1_gradient, c = simple1_constraints, x0=simple1_
  init, n=2000),
53         "simple2" => (f=simple2, g=simp
  le2_gradient, c = simple2_constraints, x0=simple2_
  init, n=2000),
54         "simple3" => (f=simple3, g=simp
  le3_gradient, c = simple3_constraints, x0=simple3_
  init, n=2000))
55
56
57
58 """
59     count(f::Function)
60     count(f, g)
61     count(f, g, c)
62
63 Check how many times the function f has been calle
  d, or calculate `f + 2g`, or `f + 2g + c`
64 """
65 Base.count(f::Function) = get(COUNTERS, string(nam
  eof(f)), 0)
66 Base.count(f::Function, g::Function) = count(f) +
  2*count(g)
67 Base.count(f::Function, g::Function, c::Function)
  = count(f) + 2*count(g) + count(c)
68
69
70 """
71     get_score(f, g, c, x, n)
72
73 The score is computed as `f(x)` using the potenti
  al optimum `x`.
74 If `count(f, g) + count(c) > n`, or the constraint
  s are violated, the score is increased significant
  ly,
75 with overevaluating being penalized much harsher t
  han constraint violation.
76 Also returns the number of evaluations.
77 """
78 function get_score(f, g, c, x, n)
79     num_evals = count(f, g) + count(c)
80
81     # helper function to compute the inf-norm pena
  lty
82     p_max(x) = max(maximum(c(x)), 0)
83
84     score = f(x)
85     score += (num_evals>n)*1e9 + (p_max(x)>0)*1e7
86
87     return num_evals, score
88 end
89
90
91 """
92     main(probname, repeat, opt_func)
93
94 Evaluates a problem given by `probname` `repeat` t

```

```

41 end
42
43
44 # simple.jl defines the 3 simple problems. It's in
  cluded down here rather than at the
45 # top because it relies on @counted and COUNTERS t
  o track the evaluation counts.
46 include("simple.jl")
47
48 const PROBS = Dict{"simple1" => (f=simple1, g=simp
  le1_gradient, c = simple1_constraints, x0=simple1_
  init, n=2000),
49         "simple2" => (f=simple2, g=simp
  le2_gradient, c = simple2_constraints, x0=simple2_
  init, n=2000),
50         "simple3" => (f=simple3, g=simp
  le3_gradient, c = simple3_constraints, x0=simple3_
  init, n=2000))
51
52
53
54 """
55     count(f::Function)
56     count(f, g)
57     count(f, g, c)
58
59 Check how many times the function f has been calle
  d, or calculate `f + 2g`, or `f + 2g + c`
60 """
61 Base.count(f::Function) = get(COUNTERS, string(nam
  eof(f)), 0)
62 Base.count(f::Function, g::Function) = count(f) +
  2*count(g)
63 Base.count(f::Function, g::Function, c::Function)
  = count(f) + 2*count(g) + count(c)
64
65
66 """
67     get_score(f, g, c, x, n)
68
69 The score is computed as `f(x)` using the potenti
  al optimum `x`.
70 If `count(f, g) + count(c) > n`, or the constraint
  s are violated, the score is increased significant
  ly,
71 with overevaluating being penalized much harsher t
  han constraint violation.
72 Also returns the number of evaluations.
73 """
74 function get_score(f, g, c, x, n)
75     num_evals = count(f, g) + count(c)
76
77     # helper function to compute the inf-norm pena
  lty
78     p_max(x) = max(maximum(c(x)), 0)
79
80     score = f(x)
81     score += (num_evals>n)*1e9 + (p_max(x)>0)*1e7
82
83     return num_evals, score
84 end
85
86 """
87     main(probname, repeat, opt_func)
88
89 Evaluates a problem given by `probname` `repeat` t

```

```

    times using `opt_func`
95 as the optimization (pass in your `optimize`). Ret
    urns the number of evaluations
96 for each trial and each trial's score.
97
98 ## Arguments:
99   - `probname`: Name of optimization problem (e.
    g. "simple1")
100   - `repeat`: Number of Monte Carlo evaluations
101   - `opt_func`: Optimization algorithm
102 ## Returns:
103   - (`scores`, `nevals`)
104 """"
105 function main(probname::String, repeat::Int, opt_f
    unc, seed = 42)
106     prob = PROBS[probname]
107     f, g, c, x0, n = prob.f, prob.g, prob.c, prob.
    x0, prob.n
108     scores = zeros(repeat)
109     nevals = zeros(Int, repeat)
110     optima = Vector{typeof(x0())}(undef, repeat)
111     # Repeat the optimization with a different ini
    tialization
112     for i in 1:repeat
113         empty!(COUNTERS) # fresh eval-count each t
    ime
114         Random.seed!(seed + i)
115         optima[i] = opt_func(f, g, c, x0(), n, pro
    bname)
116         nevals[i], scores[i] = get_score(f, g, c,
    optima[i], n)
117     end
118     return scores, nevals, optima
119 end
120
121
122
123

```

```

    times using `opt_func`
88 as the optimization (pass in your `optimize`). Ret
    urns the number of evaluations
89 for each trial and each trial's score.
90
91 ## Arguments:
92   - `probname`: Name of optimization problem (e.
    g. "simple1")
93   - `repeat`: Number of Monte Carlo evaluations
94   - `opt_func`: Optimization algorithm
95 ## Returns:
96   - (`scores`, `nevals`)
97 """"
98 function main(probname::String, repeat::Int, opt_f
    unc, seed = 42)
99     prob = PROBS[probname]
100     f, g, c, x0, n = prob.f, prob.g, prob.c, prob.
    x0, prob.n
101     scores = zeros(repeat)
102     nevals = zeros(Int, repeat)
103     optima = Vector{typeof(x0())}(undef, repeat)
104     # Repeat the optimization with a different ini
    tialization
105     for i in 1:repeat
106         empty!(COUNTERS) # fresh eval-count each t
    ime
107         Random.seed!(seed + i)
108         optima[i] = opt_func(f, g, c, x0(), n, pro
    bname)
109         nevals[i], scores[i] = get_score(f, g, c,
    optima[i], n)
110     end
111     return scores, nevals, optima
112
113
114
115

```



ads via Carbon

Develop FaaS than  
 eve  
 FaaS & DBaaS to  
 deliver the power of  
 autoscaling. Get \$300  
 in credit