123 lines - 9 Removals

```

"""
    helpers.jl -- Feel free to modify anything you like in this file!

Provides the problem structure and some useful functions. In particular, students may be
interested in `count` - to help with writing their algorithm, and `main` - to test their
methods. Note that `starter_code/localtest.jl` provides an identical test to that found
in the autograder. `main`, on the other hand, returns other metrics that may be of
interest during development.
"""

# Statistics is the only allowed external package (Random is part of the standard library)
# If you would like to use any other (standard) packages, make sure to do so in `project2.jl`
# *NOT HERE*.
using Random
using Statistics

# A global counter that keeps track of how many times each function has been called.
# It may seem like a clever hack to edit this dictionary as part of your optimize
# method to get infinite function evaluations, but beware...
# you'll regret it when it goes through the autograder
const COUNTERS = Dict{String, Int}()

"""
    @counted

A function defined with this macro increments the global counter `COUNTERS`
each time it's called.

Example:
    @counted f(x) = 2x  #each time `f(x)` is called, we also have `COUNTERS["f"] += 1`
"""
macro counted(f)
    name = f.args[1].args[1]
    name_str = String(name)
    body = f.args[2]
    update_counter = quote
        if !haskey(COUNTERS, $name_str)
            COUNTERS[$name_str] = 0
        end
        COUNTERS[$name_str] += 1
    end
    insert!(body.args, 1, update_counter)
    return f
```

114 lines 0 Additions

```
"""
    helpers.jl -- Feel free to modify anything you like in this file!
Provides the problem structure and some useful functions. In particular, students may be
interested in `count` - to help with writing their algorithm, and `main` - to test their
methods. Note that `starter_code/localtest.jl` provides an identical test to that found
in the autograder. `main`, on the other hand, returns other metrics that may be of
interest during development.
"""

# Statistics is the only allowed external package (Random is part of the standard library)
# If you would like to use any other (standard) packages, make sure to do so in `project2.jl`
# *NOT HERE*.
using Random
using Statistics

# A global counter that keeps track of how many times each function has been called.
# It may seem like a clever hack to edit this dictionary as part of your optimize
# method to get infinite function evaluations, but beware...
# you'll regret it when it goes through the autograder
const COUNTERS = Dict{String, Int}()

"""
    @counted
A function defined with this macro increments the global counter `COUNTERS`
each time it's called.
Example:
    @counted f(x) = 2x  #each time `f(x)` is called, we also have `COUNTERS["f"] += 1`
"""
macro counted(f)
    name = f.args[1].args[1]
    name_str = String(name)
    body = f.args[2]
    update_counter = quote
        if !haskey(COUNTERS, $name_str)
            COUNTERS[$name_str] = 0
        end
        COUNTERS[$name_str] += 1
    end
    insert!(body.args, 1, update_counter)
    return f
```

```
end


# simple.jl defines the 3 simple problems. It's included down here rather than at the
# top because it relies on @counted and COUNTERS to track the evaluation counts.
include("simple.jl")

const PROBS = Dict("simple1" => (f=simple1, g=simple1_gradient, c = simple1_constraints, x0=simple1_init, n=2000),
                   "simple2" => (f=simple2, g=simple2_gradient, c = simple2_constraints, x0=simple2_init, n=2000),
                   "simple3" => (f=simple3, g=simple3_gradient, c = simple3_constraints, x0=simple3_init, n=2000))



"""
    count(f::Function)
    count(f, g)
    count(f, g, c)

Check how many times the function f has been called, or calculate `f + 2g`, or `f + 2g + c`
"""
Base.count(f::Function) = get(COUNTERS, string(nameof(f)), 0)
Base.count(f::Function, g::Function) = count(f) + 2*count(g)
Base.count(f::Function, g::Function, c::Function) = count(f) + 2*count(g) + count(c)


"""
    get_score(f, g, c, x, n)

The score is computed as `f(x*)` using the potential optimum `x`.
If `count(f, g) + count(c) > n`, or the constraints are violated, the score is increased significantly,
with overevaluating being penalized much harsher than constraint violation.
Also returns the number of evaluations.
"""
function get_score(f, g, c, x, n)
    num_evals = count(f, g) + count(c)

    # helper function to compute the inf-norm penalty
    p_max(x) = max(maximum(c(x)), 0)

    score = f(x)
    score += (num_evals>n)*1e9 + (p_max(x)>0)*1e7

    return num_evals, score
end


"""
    main(probname, repeat, opt_func)

Evaluates a problem given by `probname` `repeat` t
```

```
end


# simple.jl defines the 3 simple problems. It's included down here rather than at the
# top because it relies on @counted and COUNTERS to track the evaluation counts.
include("simple.jl")

const PROBS = Dict("simple1" => (f=simple1, g=simple1_gradient, c = simple1_constraints, x0=simple1_init, n=2000),
                   "simple2" => (f=simple2, g=simple2_gradient, c = simple2_constraints, x0=simple2_init, n=2000),
                   "simple3" => (f=simple3, g=simple3_gradient, c = simple3_constraints, x0=simple3_init, n=2000))



"""
    count(f::Function)
    count(f, g)
    count(f, g, c)
Check how many times the function f has been called, or calculate `f + 2g`, or `f + 2g + c`
"""
Base.count(f::Function) = get(COUNTERS, string(nameof(f)), 0)
Base.count(f::Function, g::Function) = count(f) + 2*count(g)
Base.count(f::Function, g::Function, c::Function) = count(f) + 2*count(g) + count(c)


"""
    get_score(f, g, c, x, n)
The score is computed as `f(x*)` using the potential optimum `x`.
If `count(f, g) + count(c) > n`, or the constraints are violated, the score is increased significantly,
with overevaluating being penalized much harsher than constraint violation.
Also returns the number of evaluations.
"""
function get_score(f, g, c, x, n)
    num_evals = count(f, g) + count(c)

    # helper function to compute the inf-norm penalty
    p_max(x) = max(maximum(c(x)), 0)

    score = f(x)
    score += (num_evals>n)*1e9 + (p_max(x)>0)*1e7

    return num_evals, score
end


"""
    main(probname, repeat, opt_func)
Evaluates a problem given by `probname` `repeat` t
```

```
imes using `opt_func`
as the optimization (pass in your `optimize`). Returns the number of evaluations
for each trial and each trial's score.

## Arguments:
    - `probname`: Name of optimization problem (e.g. "simple1")
    - `repeat`: Number of Monte Carlo evaluations
    - `opt_func`: Optimization algorithm
## Returns:
    - (`scores`, `nevals`)
"""
function main(probname::String, repeat::Int, opt_func, seed = 42)
    prob = PROBS[probname]
    f, g, c, x0, n = prob.f, prob.g, prob.c, prob.x0, prob.n

    scores = zeros(repeat)
    nevals = zeros(Int, repeat)
    optima = Vector{typeof(x0())}(undef, repeat)

    # Repeat the optimization with a different initialization
    for i in 1:repeat
        empty!(COUNTERS) # fresh eval-count each time
        Random.seed!(seed + i)
        optima[i] = opt_func(f, g, c, x0(), n, probname)
        nevals[i], scores[i] = get_score(f, g, c, optima[i], n)
    end

    return scores, nevals, optima
end

```

```
imes using `opt_func`
as the optimization (pass in your `optimize`). Returns the number of evaluations
for each trial and each trial's score.
## Arguments:
    - `probname`: Name of optimization problem (e.g. "simple1")
    - `repeat`: Number of Monte Carlo evaluations
    - `opt_func`: Optimization algorithm
## Returns:
    - (`scores`, `nevals`)
"""
function main(probname::String, repeat::Int, opt_func, seed = 42)
    prob = PROBS[probname]
    f, g, c, x0, n = prob.f, prob.g, prob.c, prob.x0, prob.n

    scores = zeros(repeat)
    nevals = zeros(Int, repeat)
    optima = Vector{typeof(x0())}(undef, repeat)

    # Repeat the optimization with a different initialization
    for i in 1:repeat
        empty!(COUNTERS) # fresh eval-count each time
        Random.seed!(seed + i)
        optima[i] = opt_func(f, g, c, x0(), n, probname)
        nevals[i], scores[i] = get_score(f, g, c, optima[i], n)
    end

    return scores, nevals, optima
end
```