The code below is a companion to the paper:

> J. Chakravorty, J. Subramanian, A. Mahajan, "Stochastic approximation based methods for computing the optimal thresholds in remote-state estimation with packet drops," in proceedings of the American Control Conference, Seattle, WA, 2017.

Introduction

The code below performs the following calculations. Consider a first-order auto-regressive process $\{E_t\}_{t\geq 0}$ which evolves as follows:

$$E_{t+1} = aE_t + W_t,$$

where a is a real number and $\{W_t\}_{t\geq 0}$ is a i.i.d. process with symmetric and unimodal distribution. In our simulations, we restrict attention to $a = 1$ and W_t distributed according to zero-mean Gaussian distribution with unit variance. These values are hard-coded in the `nextState` function defined below.

```
@inline function nextState(E)
  a = 1.0
  σ = 1.0
  a*E + σ*randn()
end
```

Whenever $|E_t| < k$, where k is a threshold that can be tuned, we incur a distortion $d(E_t)$. We hard-code $d(e) = e^2$ in the `distortion` function defined below.

```
@inline distortion(E) = E^2
```

Let S_t denote the state of the channel. It is assumed that $\{S_t\}_{t\geq 0}$ is an i.i.d. Bernoulli process with $\Pr(S_t = 0) = p_d$.

Sampling functions

Let τ denote the stopping time when $\{|E_t| \geq k\}$ and $\{S_t = 1\}$. The code below performs the following sample-path calculations.

$$L = \sum_{t=0}^{\tau-1} \beta^t d(E_t), \quad M = \sum_{t=0}^{\tau-1} \beta^t, \quad K = \sum_{t=0}^{\tau} \beta^t U_t,$$

where β is the discount factor and $U_t = \mathbb{I}\{|E_t| \geq k\}$.

Since the calculation is stochastic, there is a positive probability that τ is a big number, which can slow down the calculations. So, we set a bound `maxIterations` on the maximum size of τ. We set the default value of `maxIterations` as `10_000`. This number may need to be increased when computing the performance for large `threshold`.

```
@inline function sample(threshold, discount, dropProb; maxIterations = 10_000)

    E, L, M, K = 0.0, 0.0, 0.0, 0.0

    count = 0
    scale = 1.0

    while count <= maxIterations
        channel_on = rand() > dropProb
        transmit   = !(-threshold < E < threshold)
        success    = transmit && channel_on

        if !success
          L += scale * distortion(E)
          K += scale * transmit
          M += scale
        else
          K += scale * transmit
          # Note that we could have written
          #     k += scale
          # because `transmit` is true in this branch of the code.
          break
        end

        scale *= discount
        E = nextState(E)
        count += 1
    end

    (L, M, K)
end
```

Mini-batch averaging

The value obtained by one sample path is usually noisy. So, we smoothen it out by averaging over a

mini-batch. The default size of the mini-batch is `1000` iterations.

```
@inline function sample_average(threshold, discount, dropProb; iterations::Int=1000)

    ell, emm, kay = 0.0, 0.0, 0.0

    for i in 1:iterations
        L, M, K = sample(threshold, discount, dropProb)
        ell += L
        emm += M
        kay += K
    end
    ell /= iterations
    emm /= iterations
    kay /= iterations
    (ell, emm, kay)
end
```

Stochastic approximation

Stochastic gradient descent for costly communication

It has been shown in the paper that a threshold k is optimal for communication cost λ if

$$\partial_k C(k) = 0, \quad \text{where } C(k) = D(k) + \lambda N(k), \quad D(k) = \frac{L(k)}{M(k)}, \quad N(k) = \frac{K(k)}{M(k)}.$$

The function `sa_costly` computes the optimal threshold for given values of `cost`, `discount`, and `dropProb`. It uses Kiefer Wolfowitz algorithm. In particular, the gradient is calculated using finite differences:

$$\nabla L \approx \frac{1}{2c}[L(k+c) - L(k-c)].$$

By default, c is set to `0.1`. If we were writing this code for higher dimensions, we would replace Kiefer-Wolfowitz with the simultaneous perturbation (SPSA) algorithm, which is more sample efficient for higher dimensions.

The stochastic approximation iteration starts from an initial guess (the parameter `initial`). Its default value is `1.0`. This initialization could be useful if we have a reasonable guess for optimal threshold (e.g., the exact solution obtained by Fredholm integral equations for the case when `dropProb = 0`; see TAC 2017 paper for details).

It is not possible to detect convergence of stochastic approximation algorithms. So we run the algorithm for a fixed number of iterations (the parameter `iterations`, whose default value is `1_000`).

Stochastic approximation algorithms are sensitive to the choice of learning rates. We use ADAM to adapt the learning rates according to the sample path. The parameters `decay1`, `decay2`, `alpha`, and `epsilon` can be used to tune ADAM. In our experience, these should be left to their default values.

It is not possible to detect convergence of stochastic approximation algorithms. So we run the algorithm for a fixed number of iterations (the parameter `iterations`, whose default value is `1_000`).

Sometimes it is useful to visualize the estimates (of the threshold) as the algorithm is running. To do so, set debug to `true`, which will print the current estimate of the threshold after every 100 iterations.

The output of the function is a trace of the estimates of the threshold (therefore, it is a 1D array fo size `iterations`).

```
@fastmath function sa_costly(cost, discount, dropProb ;
    iterations :: Int     = 1_000,
       initial :: Float64 = 1.0,
        decay1 :: Float64 = 0.9,
        decay2 :: Float64 = 0.999,
       epsilon :: Float64 = 1e-8,
        alpha  :: Float64 = 0.01,
             c :: Float64 = 0.1,
         debug :: Bool    = false,
    )

    threshold = initial
    trace     = zeros(iterations)

    moment1 = 0.0
    moment2 = 0.0

    weight1 = decay1
    weight2 = decay2

    @inbounds for k in 1:iterations
        threshold_plus  = threshold + c
        threshold_minus = threshold - c

        L_plus , M_plus,  K_plus  = sample_average(threshold_plus,  discount, dropProb)
        L_minus, M_minus, K_minus = sample_average(threshold_minus, discount, dropProb)
```

```
        C_plus  = (L_plus  + cost*K_plus )/M_plus
        C_minus = (L_minus + cost*K_minus)/M_minus

        gradient = (C_plus - C_minus)/2c

        moment1 = decay1 * moment1 + (1 - decay1) * gradient
        moment2 = decay2 * moment2 + (1 - decay2) * gradient^2

        corrected1 = moment1/(1 - weight1)
        corrected2 = moment2/(1 - weight2)

        weight1 *= decay1
        weight2 *= decay2

        threshold_delta = corrected1 / ( sqrt(corrected2) + epsilon)

        threshold -= alpha * threshold_delta

        # The above analysis is valid for positive values of threshold, but
        # the above line can make threshold negative. Ideally, if the
        # threshold is negative, we should set it to zero. But if the
        # threshold is set to zero, the value of M in the next iteration will
        # also be zero; therefore, the C at the next instant will be infinity.
        # So, we set the minimum value of threshold to be 2c.

        threshold  = max(threshold, 2c)

        if debug && mod(k,100) == 0
          @printf("#:%8d, threshold=%0.6f\n", k, threshold)
        end

        trace[k] = threshold
    end

    return trace
end
```

Stochastic gradient descent for constrained communication

It has been shown in the paper that a threshold k is optimal for a communication rate α if

$$\alpha M(k) - K(k) = 0.$$

The function `sa_constrained` computes the optimal threshold for a given value of `rate`, `discount`, and `dropProb`. It uses Robbins-Monro algorithm. This algorithm converges fairly fast, so we do not use mini-batches to smoothen out the samples and set the learning rate at iteration n to be `alpha`$/n$, where `alpha` is set to 1, by default.

Sometimes it is useful to visualize the estimates (of the threshold) as the algorithm is running. To do so, set debug to `true`, which will print the current estimate of the threshold after every 100 iterations.

The output of the function is a trace of the estimates of the threshold (therefore, it is a 1D array fo size `iterations`).

```
@fastmath function sa_constrained(rate, discount, dropProb ;
    iterations :: Int     = 1_000,
       initial :: Float64 = 1.0,
        alpha  :: Float64 = 1.0,
         debug :: Bool    = false,
    )

    threshold = initial
    trace     = zeros(iterations)

    @inbounds for k in 1:iterations
        _, M, K = sample_average(threshold, discount, dropProb; iterations=1)

        rl         = alpha/k
        threshold -= rl*(rate * M - K)

        # As in the case of costly communication, the above line can make the
        # threshold negative, so we set its minimum value to be a small
        # positive number (0.05). Note that hard-coding this value means that
        # the code will not work correctly for really small values of the
        # communication rate, where we expect the optimal threshold to be
        # less than 0.05.

        threshold  = max(threshold, 0.05)
```

```
        if debug && mod(k,100) == 0
          @printf("#:%8d, threshold=%0.6f\n", k, threshold)
        end

        trace[k] = threshold
    end

    return trace
end
```

```
@everywhere begin
    include("algorithms.jl")
end
```

Traces from multiple runs

To visualize how well the stochastic approximation algorithms (implemented in `algorithms.jl`) work, we run them multiple times for any particular choice of parameters. Since this task is an embarrassingly parallel one, we run each trace on a separate core and then combine all the traces. This is implemented in the `generateTraces` function below. The inputs to this function are:

- `sa` : the stochastic approximation algorithm to be run. The function must have the signature:

  `sa(param, discount, dropProb; iterations=iterations)`

  `param` corresponds to `cost` for costly communication and `rate` for constrained communication.

- `param` : A parameter used by the `sa` function, corresponding to communication cost or rate.
- `discount`: The discount factor
- `dropProb`: The packet drop probability.
- `iterations`: Number of iterations

It also accepts the following optional argument

- `numRuns` : The number of runs.

The output is a 2D array of size `iterations * numRuns`, where column corresponds to a different traces.

```
function generateTraces(sa, param, discount, dropProb, iterations; numRuns = 100)
    tuples = pmap(_ -> sa(param, discount, dropProb; iterations=iterations), 1:numRuns)
    traces = zeros(iterations, numRuns)

    for run in 1:numRuns
        traces[:, run] = tuples[run]
    end

    return traces
end
```

Generating and saving results

The function `generateOutput` takes the same parameters as `generateTraces` plus an additional optional parameter: `saveRawData`, which defaults to `false`. It returns a data frame with three columns: mean value, mean + standard deviation, mean - standard deviation. These are shown in the plots included in the paper. This data is saved to a tab separated file (in the `output/` directory). The filename includes the name of stochastic approximation algorithm, and values of parameter, discount, and dropProb.

When `saveRawData` is set to `true`, the traces are saved to a `.jld` file.

```
using JLD, DataFrames

function generateOutput(sa, param, discount, dropProb, iterations;
    numRuns  = 100, saveRawData = false)

    traces = generateTraces(sa, param, discount, dropProb, iterations; numRuns = 100)
    meanValue, std = mean_and_std(traces, 2)

    meanValue = vec(meanValue)
    std       = vec(std)

    # Generate summary statistics.
    stats = DataFrame(mean=meanValue,
                      upper=meanValue + std, lower=meanValue - std)

    filename = string("output/", sa,             "__parameter_", param,
                      "__discount_" , discount,  "__dropProb_" , dropProb)

    # Save summary to TSV file
    writetable("$filename.tsv", stats, separator='\t', header = true)

    if saveRawData
        save("$filename.jld",         "param", param,
            "discount", discount,     "dropProb", dropProb,
            "iterations", iterations, "numRuns",  numRuns,
            "traces", traces,         "meanValue", meanValue,
            "std", std)
    end

    return stats
end
```

Results presented in ACC'17

We want to run the simulation code of `simulations.jl` on multiple cores. So, as the first step, we make sure we are running multi-core version of Julia.

```
addprocs(Sys.CPU_CORES)
include("simulations.jl")
```

We use Plots package with GR backend to plot the results.

```
using Plots
gr()
```

A helper function to display labels

```
function labeltext(s, param)
    text(string('$', s, "=", param, '$'), 8, :bottom, :right)
end

label_costly(param)      = labeltext("\\lambda", param)
label_constrained(param) = labeltext("\\alpha", param)
```

The function below computes the results for a particular stochastic approximation algorithm for a range of `paramValues`, `discountValues`, and `packetDropValues` . If the optional argument `savePlot` is `true`, then the results every combination of `discount` and `packetDrop` are saved to a pdf file in the `plots/` directory. The optional values `iterations` and `numRuns` determine the number of iterations and runs. The optional argument `labeltext` formats the label of each plot and `ylim` adjusts the y limits.

```
function computeTimeSeries(sa, paramValues, discountValues, dropProbValues;
                           iterations=1000, numRuns=100, savePlot=false,
                           labeltext=label_costly, ylim=:auto)

    for discount in discountValues, dropProb in dropProbValues
        result = Array(DataFrame, length(paramValues))

        for i in 1:length(paramValues)
            param = paramValues[i]
            result[i] = generateOutput(sa, param, discount, dropProb, iterations;
                                       numRuns=numRuns)
        end
```

```
        if savePlot
            plt = plot(xlabel="Iterations", ylabel="Threshold",
                       aspect_ratio=1.75, ylim=ylim)

            for i in 1:length(paramValues)
                param = paramValues[i]
                df = result[i]
                plot!(plt, df[:upper], linecolor=:lightblue, label="")
                plot!(plt, df[:lower], linecolor=:lightblue, label="",
                     fillrange=df[:upper], fillcolor=:gray, fillalpha=0.8)
                plot!(plt, df[:mean],  linecolor=:black, linewidth=2, label="")

                x=div(3iterations,4)
                y=df[:upper][x]
                label=labeltext(param)

                plot!(plt,annotations=(x,y,label))
            end

            filename = string("plots/", sa,             "__parameter_", paramValues,
                              "__discount_" , discount,  "__dropProb_" , dropProb)

            savefig("$filename.pdf")
        end
    end
end
```

First we compute the results for packet drop probability $p_d = 0$ (and separately verify these results from the values obtained from numerically solving Fredholm integral equations.

```
computeTimeSeries(sa_costly, linspace(100,700,7), [0.9,1.0], 0.0;
                  iterations=10_000, numRuns=100, savePlot=false)
computeTimeSeries(sa_constrained, linspace(0.1,0.8,8), [0.9,1.0], 0.0;
                  iterations=10_000, numRuns=100, savePlot=false)
```

Next, we compute the results for packet drop probability $p_d = 0.3$.

```
computeTimeSeries(sa_costly, linespace(100,500,3), [0.9,1.0], 0.3;
                  iterations=10_000, numRuns=100,
                  savePlot=true, labeltext=label_costly, ylim=(0,11))
computeTimeSeries(sa_constrained, linspace(0.1,0.5,3), [0.9,1.0], 0.0;
                  iterations=1000, numRuns=10,
                  savePlot=true, labeltext=label_costly, ylim=(0,3.5))
```