Directed Research - Performance Modeling

Queueing Network Model Solvers

Paul Howe

pahowe@cs.usfca.edu

Sponsored by Professor Jeff Buckwalter

buckwalter@usfca.edu

CS698 Section 63

University of San Francisco

Spring 2011

Table of contents

Queueing Network Model Solvers

1. Description

In queueing theory, a scenario is defined in terms of queueing centers, delay centers and customers.□ Customers at a bank who wait for service from a number of bank tellers. This simple scenario demonstrates an example queueing network. In this research the objects of queueing theory map to components of computing systems. Queueing servers map to CPUs, think time to measured system delay etc.

Given observational data of a certain supercomputer job run such as CPU timing values this data can be interpreted as a description of a queueing network. Once a supercomputer run is described as a queuing network, queueing network modeling solvers can be applied to the input data to derive performance characteristics of the supercomputer medium.

This directed research project was aimed at implementing a set of queueing network model solvers capable of consuming observational run data and producing system performance characteristics.

Throughout the semester we have developed two sets of queueing network model (QNM) solvers to accomplish the above goal. One set which implements a single class exact model solver and a second which implements an approximate multiple class model solver.

Single Class Exact QNM Solver

The single class exact queueing network model solver is capable of providing exact solutions to QNM problems. Problems are described in a configuration format (described below) which specifies the inputs to□ the solver (such as center values, delay values etc). The input parameters are used to by the solving algorithm to yield an exact solution which contains outputs for different properties of the solved network. A detailed description of the output values is included below.

Because this solver yields exact solutions it is much more computationally expensive than the approximate solver counterpart. The exact solver is useful for networks of small to medium size. It is also valuable as a tool for testing and verifying correctness of problem solutions. The single class and multiple class solver outputs may be compared to determine the relative accuracy of solutions from the approximate solvers.

Multiple Class Approximate QNM Solver

The multiple class approximate solver is a more versatile counterpart to the single class exact MVA solver. Because it relies on finding approximations rather than exact solutions it is able to solve scenarios of sizes□ out of reach of the exact solver.

The multiple class approximate QNM solver produces outputs of the same format as the single class exact solver. This is intended so that comparison between both types is straightforward.

2. Purpose

Our goals for this directed research study are to improve the nature of supercomputer modeling research in the context of queuing theory. To achieve this goal we have pursued the development of an array of supporting tools and infrastructure to create a platform for improved research.

The tools built to support this research include a set of queuing network model mean value analysis solvers, an observational results database as well as data loading and processing tools for manipulating raw results data.

3. Input

Solver input is specified in a configuration file. The configuration format defines solver scenarios. The scenarios describe the queueing network model inputs such as the number of customers in the system, the queueing center times etc.

Configuration format

The configuration used to describe the queueing networks is formatted as a YAML document. YAML is a simple markup language in use in a variety of contexts. YAML is a convenient format for this type of usage for several reasons. First YAML documents are self documenting. Significant whitespace provides structure to data values and there is a minimum of syntax. Describing scenarios in YAML not only yields a stable readable representation of solver scenarios but it also serves as a representation that is directly consumable by our solver implementations.

The configuration format also supports comments which gives the opportunity to add further descriptions to clarify the nature of the scenario. There are also YAML parsers for practically every significant language which makes sharing the same format among systems developed in different languages quite easy. Because of the maturity of these different parsers were able to receive a representation of the input data that is more or less stable among all languages which is an achievement considering that different languages may have different primitive data types and mapping types from one to another may not always be obvious (ex. R vectors to Python lists or tuples).

The last feature of this format which has been proven useful is only definitions relevant to the implementations are considered. This means that the format can be extended without affecting any systems which consume information from these documents because values not relevant to a given consumer are simply ignored.

Configuration format structure

The configuration format groups problem data into a hierarchical structure. At the top level parameters global to the entire problem are defined. For example a **`tolerance`** value may be defined at the top level because it is a global value which dictates how far the calculation should proceed when using the approximate QNM solver.

The most important top level item is **`class info`**. The **`class info`** section contains all of the class specific problem information. Beneath this item we can define each class by name and each classes relevant input values. Detailed examples can be seen below.

Per-class information is encapsulated by a class label. The example's listed use class names such as **'A'** and **'B'** etc however you may want to use a more descriptive label such as **short-message** or any other type of label which accurately describes the character of the class.

Note on specifying center values

Center values (both queue and delay) may be specified in two ways. Single literal values are given to specify a specific value for the center corresponding to the index of that value. So for example, to specify two queueing centers with values the values '10' and '20' we can write,

```
queue center demands : [10, 20]
```

In the above we've simply defined two queueing centers. The first center has a value of 10 and the second 20. This notation is adequate when there are a small number of server values to be defined. However, when dealing with scenarios in which there is a large number of centers this quickly becomes awkward. To address this, an alternative notation is provided by which a run length encoded value may be supplied. Repeating values may be specified by providing a run length prior to the corresponding value. For example,

```
queue center demands : [ [100, 10], [500, 5] ]
```

Above we have defined a vector queuing center values of length 600. Clearly compressing repeated values with a run length greatly simplifies the input specification.

Finally, for ultimate control one may mix both forms to specify centers values.

```
queue center demands : [ 5, 10, [100, 10], [500, 5], 20, 1000 ]
```

Examples

Single class

```
# 160 customers, 100 centers @ 1.0

class info:

  A:
    customers: 160
    queue center demands: [[100, 1.0]]
    delay center demands: []

tolerance: 10E-4
```

Here we define a scenario which has a single class and 160 customers. In addition there are 100 queue centers which each have a value of 1.0. We also set a tolerance value of 0.0001. This is needed for the approximate solver and will be ignored by the exact solver.

Multiple classes

```
# 101 centers, 80 customers in each class

class info:

  A:
    customers: 80
    queue center demands: [[100, 1.0]]
    delay center demands: [0]

  B:
    customers: 80
    queue center demands: [[100, 1.0]]
    delay center demands: [1000]

tolerance: 10E-4
```

In this example we define two classes `A` and `B`. Each class has 101 centers, 100 queueing centers and a single delay center. Both are set to 80 customers.

Note that though we have defined two classes this scenario can still be consumed by our single class exact solver. The single class exact solver will simply discard all class definitions after the first entry.

4. Output

Format description

Solver results are printed in a consistently for all implementations and algorithms. Results are grouped by class and each class' output contains two types of values. The first is class level values such as throughput□ which apply to the entire class. The second are center related values such as utilization. For values such as utilization, each center may have a unique value therefore centers are enumerated and printed with their corresponding output values.

Result types

- Class level: Throughput, response time
- Center level: Queue lengths, utilization, residence times

In the way that input values may be specified in a compressed run length encoded format, multi-value output□ is always printed in run length encoded form. This is done so that solver results of scenarios which have many centers may reasonably fit in as few screens of output as possible.□

The following examples correspond to the previously described input scenarios.

Single class output example

```
class:  A

    throughput: 6.178e-01
    response time: 2.590e+02

    queue lengths
    100 x 1.600e+00

    utilization
    100 x 6.178e-01

    residence times
    100 x 2.590e+00
```

Multiple class output example

```
class:  A

    throughput: 4.168e-01
    response time: 1.919e+02

    queue lengths
    100 x 8.000e-01
    1 x 0.000e+00

    utilization
    100 x 4.168e-01
    1 x 0.000e+00

    residence times
    100 x 1.919e+00
    1 x 0.000e+00

class:  B

    throughput: 6.707e-02
    response time: 1.193e+03

    queue lengths
    100 x 1.293e-01
    1 x 6.707e+01

    utilization
    100 x 6.707e-02
    1 x 6.707e+01

    residence times
    100 x 1.928e+00
    1 x 1.000e+03
```

Notice in the example output above that repeated values are only printed once. We see that for one run, all 100 centers have the same queue length of 1.600e+00. This compressed form of output enables us to solve scenarios with large numbers of centers while providing output which is of a manageable length.

5. Process

Algorithmic definition

Equation 1: Little's law applied to the entire queueing network

$$X_c(\vec{N}) = \frac{N_c}{Z_c + \sum_{k=1}^{K} R_{c,k}(\vec{N})}$$

Equation 2: For each class, Little's law applied to each center

$$Q_{c,k}(\vec{N}) = X_c(\vec{N}) R_{c,k}(\vec{N})$$

Equation 3: Determining queue lengths at each center

$$Q_k(\vec{N}) = \sum_{c=1}^{C} Q_{c,k}(\vec{N})$$

Equation 4: For each class, residence time at each center

$$Q_{c,k} = \begin{cases} D_{c,k} & (delay\, centers) \\ D_{c,k}\,[1 + A_{c,k}(\vec{N})] & (queuing\, centers) \end{cases}$$

Single class exact algorithm pseudo-code

```
gather inputs from scenario file
initialize queue lengths to zero for all centers

for each customer, n

  for each center, k
    compute the residence time at the center k

  using little's law, compute throughput

  for each center, k
    compute the queue length of center k

  for each center, k
    compute utilization at center k

print results
```

Multiple class approximate algorithm pseudo-code

```
gather inputs from scenario file

initialize queue lengths with initial approximation:
  q[c] = number of customers in class c / number of centers

while delta is greater than the tolerance threshold begin

  # establish new queue length approximations
  for each class, c
    for each center, k
      set average[(c, k)] = (( number of customers in class c - 1) * q[(c,k)]) + sum(q[(j,k)])

  # using equations 7.1, 7.2, 7.3, estimate new network values

  # residence time
  for each class, c:
    for each center, k:

      if delay center:
        set residence_time[(c,k)] = delay center demand
      else:
        set residence_time[(c,k)] = center demand * (1 + average[(c,k)])

  # throughput
  for each class, c:
    for each center, k:
      set throughput[(c,k)] = number of customers in class c / sum(residence_time[(c,k)])

  # queue lengths
  for each class, c:
    for each center, k:
      set new q[(c,k)] = throughput[c] * residence_time[(c,k)]

  # utilization
  for each class, c:
    for each center, k:
      set utilization[(c,k)] = throughput[c] * center demand[(c,k)]

  set old queue lengths to new queue lengths
  determine iteration delta

end while

print results
```

6. Execution

How to execute

The following describes how to run of each the solvers for each implementation (Python or R).

Examples

The solvers conform to the same standard for executing against a specific input scenario. Simply choose the□ solver and pass in as the first argument the desired scenario name.□

Multiple class approximate [R]

```
$ multi_class_approx.r input
```

Multiple class approximate [py]

```
$ multi_class_approx.py input
```

Single class exact [R]

```
$ single_class_exact.r input
```

Single class exact [py]

```
$ single_class_exact.py input
```

7. End Notes

The project source code is hosted in GIT on our research virtual machine `hyper.cs.usfca.edu` hosted by CS labs.

Assets

The annotated source with commentary can be found at http://hyper.cs.usfca.edu/~pahowe/mpip-docs/. The GIT repository web view can be found at http://hyper.cs.usfca.edu:1234.

8. References

Edward D. Lazowska, John Zahorjan, G. Scott Graham, and Kenneth C. Sevcik. 1984. Quantitative System Performance: Computer System Analysis Using Queueing Network Models. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Wikipedia, YAML, http://en.wikipedia.org/wiki/YAML (as of May 19, 2011, 19:41 GMT).

Wikipedia, R (programming language), http://en.wikipedia.org/wiki/R_(programming_language) (as of May 19, 2011, 19:41 GMT).

Wikipedia, Python (programming language), http://en.wikipedia.org/wiki/Python_(programming_language) (as of May 19, 2011, 19:40 GMT).

9. Appendix (Source code)

Utilities

R language utilities

```
#!/usr/bin/env Rscript

## ### Base utils
## This file contains supporting functions for our R based solvers, MVAs etc

## Shorthand paste, useful for making index pseudo-keys
## i.e. foo[p(1,2)] is the same as foo["1 2"]
p <- paste

## Simple wrapper to provide C like printf
printf <- function(fmt, ...){ writeLines(sprintf(fmt, ...)) }

encode_run_length <- function(val_list){

  acc <- c()
  count <- 1
  prev_val <- NULL

  for(key in 1:length(val_list)){
    val <- val_list[[key]];

    if(!is.null(prev_val) && (val == prev_val)){
      count <- count + 1
    }else if(!is.null(prev_val)){
      acc <- c(acc, p(count, prev_val))
      count <- 1
    }

    prev_val <- val


  }

  acc <- c(acc, p(count, val))
  acc

}

decode_run_length <- function(rle_list){

  expanded <- c()

  if(length(rle_list) < 1){
    return(expanded)
  }

  for(i in 1:length(rle_list)){
    item <- rle_list[[i]]

    # run length, value pair
    if(length(item) > 1){

      run_length <- item[[1]]
      val <- item[[2]]

    # single item, just tack on
    }else{

      run_length <- 1
      val <- item

    }
    # expand rle pair
    for(k in 1:run_length){
      expanded <- c(expanded, val)
    }
  }

  expanded
}
```

Python language utilities

```
# Takes a list of tuples of type (index, value)
# and returns a run length compressed / encoded list of values
def encode_run_length(val_list):

  acc = []
  count = 1
  prev_val = None

  for key, val in val_list:

    if val == prev_val:
      count += 1
    elif prev_val is not None:
      acc.append((count, prev_val))
      count = 1

    prev_val = val

  acc.append((count, val))

  return acc

def decode_run_length(rle_list):

  # expand any run length encoded values
  # demand is either a single value or a tuple like: (run length, value)
  expanded = []
  for item in rle_list:

    normal_item = isinstance(item, list) and item or [item]
    normal_item.insert(0, 1)

    # take last 2 items
    run_length, val = normal_item[-2:]

    for i in range(run_length):
      expanded.append(val)

  return expanded
```

R single class exact MVA solver

```
#!/usr/bin/env Rscript

## ### Implementation of Single Class Exact MVA
## from QSP, appendix A



library(yaml)

source('utils.r')

single_class_exact_mva <- function(num_centers,
                                   num_customers,
                                   demand,
                                   class_name){

  qlen  <- vector('numeric', length=num_centers)
  rtime <- vector('numeric', length=num_centers)
  tput <- 0
  sysr <- 0

  for(n in 1:num_customers){

    sysr <- 0

    for(center in 1:num_centers){

      rtime[center] = demand[center] * (1.0 + qlen[center])
      sysr <- sysr + rtime[center]
    }

    residence_sum = 0
    for(k in 1:num_centers){
      residence_sum <- residence_sum + rtime[k]
    }
    tput <- n / (residence_sum)

    for(center in 1:num_centers){
      qlen[center] <- rtime[center] * tput
    }

  }


  printf('class: %s', class_name)
  printf('\n\tthroughput: %s', tput)
  printf('\tresponse time: %s\n', (num_customers / tput))

  encoded_qlen <- encode_run_length(qlen)
  printf('\tqueue lengths')
  for(i in 1:length(encoded_qlen)){
    printf('\t%s', encoded_qlen[i])
  }

  ## reckon utilization values
  utilization <- vector('numeric', length=num_centers)
  for(center in 1:num_centers){
    utilization[center] <- (tput * demand[center])
  }

  encoded_utilization <- encode_run_length(utilization)
  printf('\n\tutilization')
  for(i in 1:length(encoded_utilization)){
    printf('\t%s', encoded_utilization[i])
  }

  encoded_rtime <- encode_run_length(rtime)
  printf('\n\tresidence times')
  for(i in 1:length(encoded_rtime)){
    printf('\t%s', encoded_rtime[i])
  }
  printf('')

}

args <- commandArgs()
scenario_fn <- args[6]
scenario <- yaml.load_file(scenario_fn)

class_info = scenario[['class info']][[1]]
class_name = names(scenario[['class info']])[[1]]

num_centers <- length(class_info[['queue center demands']])
num_customers <- class_info[['customers']]

demand <- decode_run_length( class_info[['queue center demands']] )

single_class_exact_mva(num_centers=num_centers,
                       num_customers=num_customers,
                       demand=demand,
                       class_name=class_name)
```

Python single class exact MVA solver

```
#!/usr/bin/env python

# Implementation of Single Class Exact MVA
# from QSP, appendix A

import sys
import yaml
import random

from utils import encode_run_length
from utils import decode_run_length

# Setup vars
# <pre>
#    integer Ncusts,Ncents,n,center
#    real demand(25)
#    real qlen(25)
#    real rtime(25)
#    real tput, sysr
# </pre>

# Solve for example in QSP: P. 117

# load scenario from file

if len(sys.argv) < 2:
  print 'usage: $ ./single_class_exact.py scenario'
  sys.exit()

scenario_name = sys.argv[-1]
scenario = yaml.load(file(scenario_name).read())

# assume that first class defined is the class were
# interested, ignore any others that may be defined
# later

class_name, class_info = scenario['class info'].items()[0]
num_centers = len(class_info['queue center demands'])

#queue_centers = class_info['queue center demands']
#delay_centers = class_info['delay center demands']

num_customers = class_info['customers']

rtime = dict(zip(range(num_centers+1), [0 for x in range(num_centers+1)]))

tput = 0
sysr = 0

# prepend w/ zero as algorithm is index one based
demand = [[1, 0]] + class_info['queue center demands']
# expand RLE encoded items
demand = decode_run_length(demand)

# Gather inputs
# <pre>
#     write (6,5)
# 5   format (27h Input number of customers:)
#     read (5,101 Ncusts
# 10  format (i4)
#     write (6,15)
# 15  format (25h Input number of centers:)
#     read (5,10) Ncents
#     write (6,20)
# 20  format (25h Input service demand for)
#
#     do 25 center = 1, Ncents
#         write (6,30) center
# 30      format (IOh Center ,i2,lh:)
#         read (5,351 demand(center)
# 35      format(f8.4)
# 25      continue
# </pre>

# Now that the network is described, we perform the evaluation.
# Begin by initializing the trivial solution for zero customers.
# <pre>
#     do 40 center = 1,Ncents
#         qlen(center) = 0.0
# 40      continue
# </pre>
qlen = dict(zip(range(num_centers+1), [0 for x in range(num_centers+1)]))

# The algorithm solves successively for each population.
# <pre>
#     do 45 n = 1,Ncusts
#
#         c First, compute the residence time at each center.
#
#         sysr = 0.0
#         do 50 center = 1,Ncents
# 2001        rtime (center) = demand(center) * (1.0 + qlen(center))
#             sysr = sysr + rtime(center)
# 50          continue
# </pre>
for n in range(1, num_customers+1):

  # compute residence time at each center
  sysr = 0

  for center in range(1, num_centers+1):

    rtime[center] = demand[center] * (1.0 + qlen[center])
    sysr = sysr + rtime[center]

  # Next, use little's law to compute system throughput.
  # <pre>
  #         tput = n / sysr
  #         do 55 center = 1,Ncents
  # 2003        qlen(center) = rtime(center) * tput
  # 55          continue
  # c
  # 45      continue
  # </pre>
  tput = n / sum([rtime[k] for k in range(1, num_centers+1)])

  for center in range(1, num_centers+1):
    qlen[center] = rtime[center] * tput

#  Print results.
# <pre>
#     write (6,60) tput
# 60  format (20h System throughput:, f8.4)
#     write(6,65) Ncusts/tput
# 65  format (23h System response time: ,f8.4)
# </pre>
#print '\nSystem throughput: ', tput
#print 'System response time: ', (num_customers / tput) - think_time

# <pre>
#     write (6,70)
# 70  format (22h Device utilizations: )
#     do 75 center=1, Ncents
#         write (6,80) center, tput*demand(center)
# 80      format (i5, 2h: ,f5.3)
# 75      continue
# </pre>

#print '\nDevice utilizations: '
utilization = {}
for center in range(1, num_centers+1):
#  print 'center: %s, %s' % (center, tput * demand[center])
  utilization[center] = tput * demand[center]
```

```
# <pre>
#     write (6,85)
# 85  format (23h Device queue lengths: )
#     do 90 center=1,Ncents
#       write (6,95) center,qlen(center)
# 95    format (i5,2h: ,f8.4)
# 90    continue
#       end
# </pre>
#print '\nDevice queue lengths: '
#for center in range(1, num_centers+1):
#  print 'center: %s, %s' % (center, qlen[center])

#print '\nDevice residence times: '
#for center in range(1, num_centers+1):
#  print 'center: %s, %s' % (center, rtime[center])

classes = [class_name]

sorted_utilizations = sorted(utilization.items())
encoded_utilizations = encode_run_length(sorted_utilizations)

sorted_qlengths =  sorted(qlen.items())[1:]
encoded_qlengths = encode_run_length(sorted_qlengths)

sorted_res_times = sorted(rtime.items())[1:]
encoded_res_times = encode_run_length(sorted_res_times)

for c in classes:
  print 'class: ', c, '\n'

  print '\tthroughput: ', tput
  print '\tresponse time: ', (num_customers / tput), '\n'

  print '\tqueue lengths'
  for count, val in encoded_qlengths:
    print '\t', '%s x %s' % (count, val)
  print ''

  print '\tutilization'
  for count, val in encoded_utilizations:
    print '\t', '%s x %s' % (count, val)
  print
  print '\tresidence times'
  for count, val in encoded_res_times:
    print '\t', '%s x %s' % (count, val)
```

R Multi class approximate MVA solver

```
#!/usr/bin/env Rscript

## ### Multiple class approximate mean value analysis solver

## Implementation of multiple class approximate mean value analysis
## from QSP, Chapter 7, pg. 143

library(yaml)

source('utils.r')

args <- commandArgs()
scenario_fn <- args[6]
scenario <- yaml.load_file(scenario_fn)

classes <- names(scenario[['class info']])

num_customers <- list()
queue_center_demands <- list()
delay_center_demands <- list()
## combination of queueing and delay centers
demand_inputs <- list()

for(i in 1:length(scenario[['class info']])){
```

```r
  class          <- names(scenario[['class info']])[[i]]

  num_cust       <- scenario[['class info']][[class]][['customers']]
  center_demands <- decode_run_length(scenario[['class info']][[class]][['queue center demands']])
  delay_demands  <- decode_run_length(scenario[['class info']][[class]][['delay center demands']])

  num_customers[[class]] <- num_cust

  # YAML parser returns numbers in scientific notation as strings,
  # coerce to doubles
  queue_center_demands[[class]] <- as.double(center_demands)
  delay_center_demands[[class]] <- as.double(delay_demands)

  if(length(queue_center_demands[[class]]) > 0){
    for(i in 1:(length(queue_center_demands[[class]]))){
      demand_inputs[[class]][[i]] <- list('q', queue_center_demands[[class]][[i]])
    }
  }

  if(length(delay_center_demands[[class]]) > 0){
    for(i in 1:(length(delay_center_demands[[class]]))){
      di <- i + length(queue_center_demands[[class]])
      demand_inputs[[class]][[di]] <- list('d', delay_center_demands[[class]][[i]])
    }
  }
}

## general inputs

## assumes homogeneous center arities
num_centers <- length(demand_inputs[[1]])

tolerance   <- scenario$tolerance
centers     <- 1:num_centers

multi_class_approx_mva <- function(classes,
                                   num_customers,
                                   centers,
                                   num_centers,
                                   demand_inputs,
                                   tolerance){

  ## list values are keyed on pseudo-key values such as ('a', 1), representing
  ## the class and the center

  num_classes <- length(classes)

  delta          <- 999
  demand         <- list()
  average        <- list()
  iteration      <- 0
  qlengths       <- list()
  throughput     <- list()
  utilization    <- list()
  residence_time <- list()

  ## NB c is a reserved word in R so use ci
  ## as class-iterator

  ## expand the demand list
  for(ci in 1:num_classes){
    for(k in 1:num_centers){
      class <- classes[ci]

      print(k)

      k_demands <- demand_inputs[[classes[ci]]]
      ## 1st index is the type (delay or queue)
      ## 2nd index is the value
      ck_demand <- k_demands[[k]]
      demand[p(class,k)] <- list(ck_demand)
    }
```

```
}

## 1, initialize queue lengths with initial guess
for(ci in 1:num_classes){
  for(k in 1:num_centers){
    class <- classes[ci]

    Nc <- num_customers[[classes[ci]]]
    qlengths[p(class,k)] <- Nc / num_centers
  }
}

while(delta >= tolerance){

  new_qlengths <- list()
  residence_time_sum <- list()

  ## 2, approximate averages A[(c,k)]
  for(ci in 1:num_classes){
    for(k in 1:num_centers){
      class <- classes[ci]

      Nc <- num_customers[[classes[ci]]]
      left <- ( ( Nc - 1) / Nc ) * qlengths[[p(class,k)]]

      acc <- c()
      for(j in 1:num_classes){
        classj <- classes[j]
        if(j != ci){
          acc <- c(acc, qlengths[[p(classj,k)]])
        }
      }
      right <- sum(acc)
      average[p(class,k)] <- left + right
    }
  }

  ## using equations 7.1, 7.2, 7.3 to estimate new network values

  ## 7.3 residence time
  for(ci in 1:num_classes){

    class <- classes[ci]
    residence_time_sum[class] <- 0

    for(k in 1:num_centers){

      demand_tuple <- demand[[p(class,k)]]
      demand_type <- demand_tuple[[1]]
      demand_value <- demand_tuple[[2]]

      ## delay center
      if(demand_type == 'd'){
        residence_time[p(class,k)] <- demand_value
      ## queueing center
      }else if(demand_type == 'q'){
        residence_time[p(class,k)] <- demand_value * (1 + average[[p(class,k)]])
      }

      ## while were here, accumulate res times accross centers
      residence_time_sum[class] <- residence_time_sum[[class]] + residence_time[[p(class,k)]]
    }
  }

  ## 7.1 throughput
  for(ci in 1:num_classes){
    class <- classes[ci]
    Nc <- num_customers[[class]]

    rsum <- 0
    for(k in 1:num_centers){
      rsum <- rsum + residence_time[[p(class,k)]]
    }
```

```
      throughput[class] <- Nc / (rsum)
    }

    ## 7.2 queue lengths
    for(ci in 1:num_classes){
      class <- classes[ci]
      for(k in 1:num_centers){
        new_qlengths[p(class,k)] <- throughput[[class]] * residence_time[[p(class,k)]]
      }
    }

    ## reckon utilizations
    for(ci in 1:num_classes){
      class <- classes[ci]
      for(k in 1:num_centers){

        demand_tuple <- demand[[p(class,k)]]
        demand_type <- demand_tuple[[1]]
        demand_value <- demand_tuple[[2]]

        utilization[p(class,k)] <- throughput[[class]] * demand_value
      }
    }

    ## reckon iteration delta
    delta <- 0
    for(ci in 1:num_classes){
      class <- classes[ci]
      for(k in 1:num_centers){
        delta <- delta + abs( qlengths[[p(class,k)]] - new_qlengths[[p(class,k)]] )
      }
    }

    ## strangely, the method specified in QSP (pg. 145) produces results which deviate
    ## slightly more from the method above. So prefer the above method for now.
    ## <code>
    ## delta = max([ abs(qlengths[(c,k)] - new_qlengths[(c,k)])  for c in classes for k in centers])
    ## </code>

    ## update queue lengths with new estimates
    qlengths <- new_qlengths

    iteration <- iteration + 1

  } # while

  ## print results

  for(ci in 1:num_classes){
    class <- classes[ci]
    printf('class: %s', class)
    printf('\n\tthroughput: %s', throughput[[class]])
    printf('\tresponse time: %s', residence_time_sum[[class]])

    printf('\n\tqueue lengths')
    ck_qlens <- c()
    for(k in 1:num_centers){
      ck_qlens <- c(ck_qlens, qlengths[[p(class,k)]])
    }
    encoded_qlens <- encode_run_length(ck_qlens)
    for(i in 1:length(encoded_qlens)){
      printf('\t%s', encoded_qlens[i])
    }

    printf('\n\tutilization')
    ck_utilization <- c()
    for(k in 1:num_centers){
      ck_utilization <- c(ck_utilization, utilization[[p(class,k)]])
    }
    encoded_utilization <- encode_run_length(ck_utilization)
    for(i in 1:length(encoded_utilization)){
      printf('\t%s', encoded_utilization[i])
    }
    printf('\n\tresidence times')
    ck_residence_time <- c()
    for(k in 1:num_centers){
      ck_residence_time <- c(ck_residence_time, residence_time[[p(class,k)]])
    }
    encoded_residence_time <- encode_run_length(ck_residence_time)
    for(i in 1:length(encoded_residence_time)){
      printf('\t%s', encoded_residence_time[i])
    }

  }
}

multi_class_approx_mva(classes, num_customers, centers, num_centers, demand_inputs, tolerance)
```

Python Multi class approximate MVA solver

```
#!/usr/bin/env python

### Multi-class approximate MVA solver implementation

# make / floating point by default
from __future__ import division

import sys
import yaml
import itertools

from pprint import pprint
from utils import encode_run_length
from utils import decode_run_length



# load scenario
scenario_name = sys.argv[-1]
scenario = yaml.load(file(scenario_name).read())

class_info = dict(map(lambda x: (x,{}),  scenario['class info'].keys()))

for c in scenario['class info']:
  qd = map(lambda q: ('q', q), decode_run_length(scenario['class info'][c]['queue center demands']))
  dd = map(lambda d: ('d', d), decode_run_length(scenario['class info'][c]['delay center demands']))
  demands = qd + dd
  class_info[c]['center_demands'] = demands
  class_info[c]['num_customers'] = scenario['class info'][c]['customers']

# prepare inputs
tolerance = float(scenario['tolerance'])
classes = class_info.keys()
# assumes uniform arity of center demands
num_centers = len(class_info[classes[0]]['center_demands'])
centers = range(1, num_centers + 1)

delta = 999
demand = {}
average = {}
iteration = 0
qlengths = {}
throughput = {}
utilization = {}
residence_time = {}

# expand demand hash
for c in classes:
  for k in range(1, len(class_info[c]['center_demands'])+1):
    demand[(c, k)] = class_info[c]['center_demands'][k-1]

# 1, initialize queue lengths with initial guess
for c in classes:
  for k in centers:
    Nc = class_info[c]['num_customers'] # num customers @ class c
```

```
      qlengths[(c,k)] = Nc / num_centers

  while delta >= tolerance:

    new_qlengths = {}
    residence_time_sum = {}

    # 2, approximate averages A[(c,k)]
    for c in classes:
      for k in centers:
        Nc = class_info[c]['num_customers']
        left = ( (Nc - 1) / Nc ) * qlengths[(c,k)]
        right = sum([qlengths[(j,k)] for j in classes if j != c])
        average[(c,k)] = left + right

    # using equations 7.1, 7.2, 7.3 to estimate new network values

    # 7.3 residence time
    for c in classes:
      for k in centers:
        demand_type, d = demand[(c,k)]
        # delay center
        if demand_type == 'd':
          residence_time[(c,k)] = d
        # queueing center
        elif demand_type == 'q':
          residence_time[(c,k)] = d * (1 + average[(c,k)])

        # while were here, accumulate res times accross centers
        # but limit system response time to queue centers
        if demand_type == 'q':
          residence_time_sum[c] = residence_time_sum.get(c, 0) + \
                                  residence_time[(c,k)]

    # 7.1 throughput
    for c in classes:
      Nc = class_info[c]['num_customers'] # num customers @ class c
      rsum = sum([residence_time[(c,k)] for k in centers])
      throughput[c] = Nc / rsum

    # 7.2 queue lengths
    for c in classes:
      for k in centers:
        new_qlengths[(c,k)] = throughput[c] * residence_time[(c,k)]

    # reckon utilizations
    for c in classes:
      for k in centers:
        utilization[(c,k)] = throughput[c] * demand[(c,k)][1]

    # reckon iteration delta
    delta = 0
    for c in classes:
      for k in centers:
        delta += abs( qlengths[(c,k)] - new_qlengths[(c,k)] )

    # strangely, the method specified in QSP (pg. 145) produces results which deviate
    # slightly more from the method above. So prefer the above method for now.
    # <code>
    # delta = max([ abs(qlengths[(c,k)] - new_qlengths[(c,k)])  for c in classes for k in centers])
    # </code>

    # update queue length w/ estimate
    qlengths = new_qlengths
    iteration += 1

  for c in classes:
    print 'class: ', c, '\n'

    print '\tthroughput: ', throughput[c]
    print '\tresponse time: ', residence_time_sum[c], '\n'

    class_qlength_keys = sorted([(cq, k) for (cq,k) in qlengths if cq == c])
```

```
class_qlength_vals = [(cq[1], qlengths[cq]) for cq in class_qlength_keys]
print '\tqueue lengths'
for ck in encode_run_length(class_qlength_vals):
  print '\t', '%s x %s' % ck
print ''

class_utilization_keys = sorted([(cu,k) for (cu,k) in utilization if cu == c])
class_utilization_vals = [(cu[1] ,utilization[cu]) for cu in class_utilization_keys]
print '\tutilization'
for cu in encode_run_length(class_utilization_vals):
  print '\t', '%s x %s' % cu
print

class_residence_time_keys = sorted([(cr,k) for (cr,k) in residence_time if cr == c])
class_residence_time_vals = [(cr[1], residence_time[cr]) for cr in class_residence_time_keys]
print '\tresidence times'
for cr in encode_run_length(class_residence_time_vals):
  print '\t', '%s x %s' % cr
print
```