DATSA Tutorial

(**D**ata **A**ggregator for **T**ime **S**eries **A**nomalies)

Shengtian Zhou, Mejbah Alam, Justin Gottschlich

Intel Labs

DATSA is a data aggregator for time series anomalies. In this tutorial, we present the intuition behind DATSA and its general goals. Next, we provide detailed steps on how it can be used to generate anomalous and non-anomalous datasets.

What is DATSA?

Data Aggregator for Time Series Anomalies

DATSA is a multivariate time series data generator. It was specifically designed to enhance anomaly detection model training, testing, and validation. Therefore, much of DATSA's software design emphasizes user control of generation of anomalous and non-anomalous[1] data.

An example dataset generated by DATSA may look like the following.

Label	**Timestamp**	**Value$_1$**	**...**	**Value$_n$**
0	1	0.0000	...	0.0000
0	2	0.0175	...	0.2000
0	3	0.0150	...	0.4500
1	4	-0.1400	...	3.0000
1	5	-2.0200	...	0.5600
0	6	0.0180	...	0.2450
...	...	...	...	...

Note that label 0 means non-anomalous entry, whereas label 1 means anomalous entry.

[1] In this tutorial, we use the words normal and non-anomalous interchangeably.

DATSA System Design

The DATSA system consists of the algorithmic system, scheduler system, and dataset generation system.

Algorithmic System

The *algorithmic system* allows users to define anomalous and non-anomalous data for a dataset. Because data can be of various forms, algorithmic control over data definition provides the freedom to realize data diversity. The algorithmic system is used to define the following system parameters: time duration, time step, respect windows, controllable stochasticity, variables, labeling, and data uniqueness. Detailed explanations for each of these parts can be found in the "Dataset Creation" section.

Scheduler System

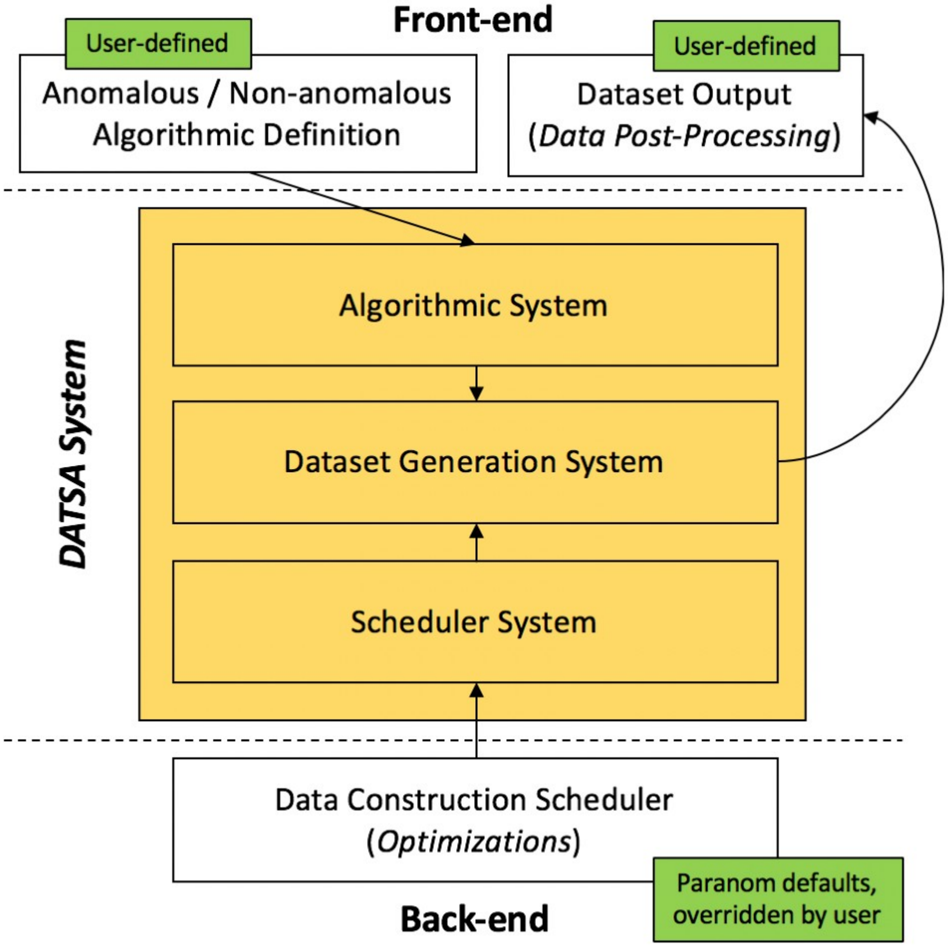
The *scheduler system* enables users to implement a scheduler for scheduling the data generation steps. It enables the user to take advantage of customized schedulers depending on different data generation constraints (e.g., size of the dataset and time for generating the dataset). Moreover, the customizability of the scheduling system enables users to optimize their dataset generation for heterogeneous and homogeneous hardware ecosystems, which can (and will) vary from user to user. The scheduler system provides a `data_generation_scheduler` class, which is the base class for any scheduler. A scheduler may extend the base class and implement the `execute` function. You can find more details about scheduler in the Open-ended Scheduler Back-end section. Also, an example scheduler implementation can be found in the Example Scheduler Back-end section.

Dataset Generation System

The dataset generation system generates multivariate time series data. It also decouples the algorithmic system and the scheduler system. An algorithm for data generation is capable of running on different back-ends, which enhances the algorithm's reusability in similar tasks with different constraints (e.g., using the same algorithm to generate a dataset of billions of data and a dataset of only ten thousand data might require different back-end platforms). The dataset generation system provides users with a `data_generator` class for constructing a dataset. For generating a dataset, the `data_generator` class requires the time duration, time step, respect windows, anomalous/non-anomalous event duration, the probability of the anomalous event anomalous/non-anomalous algorithmic definition, which are called variables, a scheduler back-end, labeling, and the specification for data uniqueness respectively. Description and example of each of these terms can be found in the Data Creation section.

System Diagram

The dataset generation system requires a scheduler and an anomalous/non-anomalous definition to construct a `data_generator` object for dataset generation. The following system diagram shows DATSA's system interactions:


Front-end
User-defined
Anomalous / Non-anomalous Algorithmic Definition
User-defined
Dataset Output (Data Post-Processing)
DATSA System
Algorithmic System
Dataset Generation System
Scheduler System
Data Construction Scheduler (Optimizations)
Paranom defaults, overridden by user
Back-end

Why DATSA?

As deep learning systems continue to grow in practical utility, richer and larger datasets will be needed to continue to improve deep learning model accuracy. In practice, the development of machine learning (ML) models generally requires iterative training and re-training with various sizes and diversity of data throughout the prototyping and deployment stages. Although data quantity may be abundant in some cases, data diversity may not be present to enhance model generalizability.

On top of the data diversity issue, *anomaly detection*, the identification of patterns in data that do not conform to expected normal behavior, presents at least two more challenges. First, in general, anomalous data is scarce. Second, anomalies tend to be continuous events.

To address these issues, we need a mechanism to generate data that is *time series* (i.e., a series of values of a quantity obtained at successive times) in nature due to the general continuous nature of anomalous events. Hence, we designed DATSA.

How to Create a Dataset?

Before creating a dataset, let's configure the environment for DATSA.

Environment Setup

DATSA is designed with a mindset of minimal software dependencies. Therefore, the only software dependency is a C++ compiler.

The GNU Compiler Collection (GCC) has been tested for DATSA.

Tested GCC version: GCC 4.2.*

Any latest version of GCC should work (other C++ compilers may work too).

DATSA Version 1.0 is supported by Unix/Linux Operating Systems.

MAC OS

If you do not have gcc installed on your mac, you can install it by typing the following command in the terminal.

```
$ xcode-select --install
```

After installation, you can check if gcc is installed successfully by typing the following commands.

```
$ gcc -v
$ make -v
```

Ubuntu

If you do not have gcc installed on your Ubuntu, you can install it by typing the following commands in the terminal.

```
$ sudo apt-get update
$ sudo apt-get upgrade
$ sudo apt-get install build-essential
```

After installation, you can check if gcc is installed successfully by typing the following commands.

```
$ gcc -v
$ make -v
```

Dataset Creation

A dataset, a time series, is composed of the following segments:

1. Time series duration
2. Non-anomalous data
3. Anomalous data

To describe these segments, the `data_generator` object is used to construct a dataset. Below is the constructor for the `data_generator`.

```
data_generator(int time_start, int time_end, int normal_lowerbound_duration,
    int normal_upperbound_duration, int anomaly_lowerbound_duration,
    int anomaly_upperbound_duration, long probability_numerator,
    long probability_denominator, int time_step = 1);
```

To generate the dataset, use the code here:

```
// data generator defined elsewhere
data_generator data_gen(...);
...

// generates a data_list object that contains the dataset
data_list list = data_gen.generate_data();
```

The dataset creation process can be classified to the following subsections: **Time Duration**, **Time Step**, **Respect Windows**, **Controllable Stochasticity**, **Variables**, **Labeling**, **Open-ended Scheduler Back-end**, and **Data Uniqueness**.

Time Duration

Each dataset starts with a starting timestamp and an ending timestamp. The timestamp is incremented by one at a time by default from start to finish. A concrete example that specifies a range from 1 to 10000 is shown below.

```
// specify a timestamp range of [1, 10000)
int time_start = 1, time_end = 10000;
```

Time Step

The timestamp may increment at a rate larger than one (i.e., the default). DATSA allows users to specify a time step *n* in the range [1, INT_MAX]. With this time step *n*, the timestamp increments by *n* at a time. In DATSA there are two ways to specify the time step.

1. Through the `data_generator` constructor
2. Through the `time_step` function in the data generator

To set the time step through the `data_generator` constructor (Note the last argument is time step):

```
// set the time step to 10
int time_step = 10;

// data generator declaration
data_generator data_gen(..., time_step);
```

To set the time step through the `time_step` function:

```
// data generator declared elsewhere
data_generator data_gen(...);

// declare the time step as 10 and set it in the data generator
int time_step = 10;
data_gen.time_step(time_step);
```

Respect Windows

A user-controlled flag `respect_windows()` can be turned on to ensure that the anomalous and non-anomalous windows are not overstepped. A user can set the size of the anomalous (or the non-anomalous) windows and the parameter `time_step` in a way that could lead to the generation of data that exceeds the specified window range in order to maintain the time steps.
When `respect_windows()` is set to true, DATSA overrides `time_step` if the `time_step` increment exceeds the window range. For example, a user can set both the upper bound and lower bound of the non-anomalous window size as 11. However, if the `time_step` is set to 5 and `respect_windows()` is turned off, the steps for a non-anomalous data would be 0, 5, 10, 15. With `respect_windows()` turned on, DATSA would instead use the time steps of 0, 5, 10, 11 to respect the user-specified window size. The default for `respect_windows()` is set to false.

The following example shows how to turn on the `respect_windows()` :

```
// data generator declared elsewhere
data_generator data_gen(...);

// respect window
data_gen.respect_windows(true);
```

The following example shows how to turn off the `respect_windows()` :

```
// data generator declared elsewhere
data_generator data_gen(...);

// disrespect window
data_gen.respect_windows(false);
```

Controllable Stochasticity

Anomalous events and normal events may have various lengths. Therefore, DATSA supports the stochastic generation of anomalous and normal durations. DATSA also requires the anomalous events' likelihood of occurrence because an anomalous event may occur with a probability.

Normal Data & Anomalous Events' Durations

The stochastic duration of a normal event or an anomalous event is specified by a lower bound and an upper bound. Stochasticity allows DATSA to randomly choose a duration from the specified range whenever a normal or an anomalous event occurs.

In the example below, a normal event's duration is from 10 to 20, whereas an anomalous event's duration is from 5 to 10.

```
// specify a stocastic normal event duration
int normal_lowerbound_duration = 10;
int normal_upperbound_duration =  20;

// specify a stocastic anomalous event duration
int anomaly_lowerbound_duration = 5;
int anomaly_upperbound_duration = 10;
```

The Probability of an Anomalous Event

The probability of an anomalous event specifies how likely an anomalous event may occur. The probability is determined by a nominator and a denominator.

In the example below, the occurring probability of an anomalous event is 0.5.

```
// specify 50% as the probability of an anomalous event
long prob_numerator = 1;
long prob_denominator = 2;
```

When the data generation process begins, DATSA decides whether the starting event will be anomalous or normal based on this probability. As in the above example, there is a 50% chance that the starting event is anomalous. If the first event is anomalous, then throughout the first duration, you will only see anomalous data. And next, DATSA again decides whether the next event will be anomalous or normal based on the same probability. And the same process goes on until the end of data generation.

Variables

Having determined the duration for anomalous and normal events and how they intertwine together, the next step is to generate values for those events.

Variables specify the values of anomalous and normal events. DATSA provides two ways of specifying value:

1. Random Variables
2. Function Variables

Once a variable is passed to the `data_generator`, the ownership of the variable is transferred, and the `data_generator` will handle variable deletion automatically.

The following two subsections explain the usage of random and function variables.

Random Variables

A random variable requires ranged normal and anomalous values.

In the univariate example below, non-anomalous data values are from 1000.0 to 2000.0 (real numbers), whereas anomalous data values are from 1.0 to 10.0 (real numbers).

```
// data generator defined elsewhere
data_generator data_gen(...);
...

// create the random variable with name "x"
random_variable *v = new  random_variable("x");

// insert anomalous range [1000.0, 1999.0] and normal range [1.0,  9.0]
v->insert_normal_range(1000.0, 2000.0);
v->insert_anomaly_range(1.0, 10.0);
```

```
  // insert the variable to the data generator
  data_gen.insert_variable(v);
```

To illustrate how to generate a multivariate anomaly dataset using random variables, we provide an example with two random variables.

```
  // data generator defined elsewhere
  data_generator data_gen(...);
  ...

  // create two random variables with name "x" and "y"
  random_variable *v1 = new random_variable("x");
  random_variable *v2 = new random_variable("y");

  // insert anomalous range and normal range
  v1->insert_normal_range(1000.0, 2000.0);
  v1->insert_anomaly_range(1.0, 10.0);
  v2->insert_normal_range(2000.0, 3000.0);
  v2->insert_anomaly_range(1.0, 20.0);

  // insert the variables to the data generator
  data_gen.insert_variable(v1);
  data_gen.insert_variable(v2);
```

Function Variables

A function variable, unlike a random variable that requires ranged values, generates the exact values for data at each timestamp.

A function variable allows value generation through user-defined algorithms for both anomalous values and normal values.

An anomalous step function (univariate) example is shown here for illustration.

```
//----------------------------------------------------------------------------
// The normal value is 1/100 of the current timestamp
//----------------------------------------------------------------------------
static double step_normal_function(event_data const &event)
{
  int step = (event.timestamp()-1);
  return double(step/100);
}

//----------------------------------------------------------------------------
// The anomalous value has the value -10.0 all the time
//----------------------------------------------------------------------------
```

```
static double step_anomaly_function(event_data const &event)
{
  return double(-10.0);
}
```

Note, the `event_data` object contains information like the current timestamp.

```

  // data generator defined elsewhere
  data_generator data_gen(...);
  ...

  // create the function variable with name "step"
  function_variable *v = new  function_variable("step");

  // refer to the functions created for value generation
  v->normal_function(&step_normal_function);
  v->anomaly_function(&step_anomaly_function);

  // insert the variable to the data generator
  data_gen.insert_variable(v);
```

To generate a multivariate dataset using a combination of function variables and random variables, we provide an example that uses two random variables and two function variables.

Normal and anomaly functions' definition for the two function variables:

```
//-----------------------------------------------------------------------------
// normal and anomaly functions for function variable v1
//-----------------------------------------------------------------------------
static double normal_function1(event_data const &event)
{
  return 1.0 * event.timestamp();
}

static double anomaly_function1(event_data const &event)
{
  return -1.0 * event.timestamp();
}

//-----------------------------------------------------------------------------
// normal and anomaly functions for function variable v2
//-----------------------------------------------------------------------------
static double normal_function2(event_data const &event)
{
  return 2.0 * event.timestamp();
}

```

```
static double anomaly_function2(event_data const &event)
{
  return -2.0 * event.timestamp();
}
```

Variables' creation and insertion:

```
  // data generator defined elsewhere
  data_generator data_gen(...);
  ...

  // declare two function variables (v1, v2) and two random variables (v3,v4)
  function_variable *v1 = new function_variable("v1");
  function_variable *v2 = new function_variable("v2");
  random_variable *v3 = new random_variable("v3");
  random_variable *v4 = new  random_variable("v4");

  // set anomaly and normal functions or range
  v1->normal_function(&normal_function1);
  v1->anomaly_function(&anomaly_function1);
  v2->normal_function(&normal_function2);
  v2->anomaly_function(&anomaly_function2);
  v3->insert_normal_range(10.0, 100.0);
  v3->insert_anomaly_range(1.0, 5.0);
  v4->insert_normal_range(20.0, 200.0);
  v4->insert_anomaly_range(2.0, 6.0);

  // insert the variables to the data generator
  data_gen.insert_variable(v1);
  data_gen.insert_variable(v2);
  data_gen.insert_variable(v3);
  data_gen.insert_variable(v4);
```

Labeling

Labels of an anomalous data or non-anomalous data are highly customizable through either the data generator or user-defined functions.

The default value is 1 for an anomaly and 0 for normality.

The below example shows an example of label customization using a data generator.

```


  // the data generator is defined elsewhere
  data_generator data_gen(...);
  ...

  // customize
```

```
data_gen.anomaly_label(0.5);
data_gen.normal_label(0.3);
```

Open-ended Scheduler Back-end

The idea of open-ended scheduler back-ends allows the separation of data generation algorithms from back-ends for efficient data generation. A scheduler back-end could be implemented however you want, and it may run on local clusters or the cloud across multiple nodes.

The intuition behind open-ended scheduler came from big data. When over billions of data are required for training a model, DATSA should deploy a scheduler back-end that makes sense concerning data generation constraints (e.g., time).

A scheduler can be implemented by extending the `data_generation_scheduler` class and implementing the `execute` function. We have provided two exemplary schedulers named `basic_scheduler` (the default scheduler) and `multithreaded_scheduler` respectively. They can be found in the datsa/src folder.

To deploy a different scheduler, declare the scheduler back-end (e.g., `multithreaded_scheduler`) and set it in the data generator:

```
// create the scheduler back-end
data_generation_scheduler *s = new multithreaded_scheduler(data_gen);

// set the scheduler back-end
data_gen.set_scheduler(s);
```

At the Example Scheduler Back-end section, we will present an example scheduler.

Data Uniqueness

Most of the real-world datasets contain a reduced number of anomalous data. Therefore, training a machine learning model with duplicates in anomalous data could potentially lead to overfitting. To address this issue, DATSA provides two data uniqueness controls, which a user might use to generate unique data.

1. *Hard uniqueness control* : A data property that must be met.
2. *Soft uniqueness control* : A data property that may be met.

Hard Uniqueness Control

If a user hard controls over a normal or anomalous event (consecutive entries of data), the data entries from that event need to have unique data values. For example, if an event consists of 4 univariate entries that have values 1, 2, 3, 3 respectively, this violates the hard uniqueness control property because 3 appeared twice in the event. When hard uniqueness is selected, if the user-defined threshold for attempts to generate a unique value is exceeded, DATSA will terminate execution. If a unique datum is generated, the total attempt count is reset, thereby ensuring each unique datum generation will be attempted `set_max_failures()` times before terminating.

An example of setting the hard uniqueness property on a normal and anomalous event is presented here:

```
// the data generator defined elsewhere
data_generator data_gen(...);
...

// set the maximum tries (100) and the hard uniqueness property
data_gen.normal_uniqueness().set_kind(kHardUniqueness);
data_gen.normal_uniqueness().set_max_failures(100);

// set the maximum tries (50) and the hard uniqueness property
data_gen.anomaly_uniqueness().set_kind(kHardUniqueness);
data_gen.anomaly_uniqueness().set_max_failures(50);
```

Soft Uniqueness Control

Similar to hard uniqueness control, soft uniqueness control tries to ensure the uniqueness over the entire dataset. However, soft uniqueness allows DATSA to continue execution if data uniqueness is not achieved after a number of user-defined tries.

An example of setting the soft uniqueness property on a normal and anomalous event is presented here:

```
// the data generator defined elsewhere
data_generator data_gen(...);
...

// set the maximum tries (100) and the soft uniqueness property
data_gen.normal_uniqueness().set_kind(kSoftUniqueness);
data_gen.normal_uniqueness().set_max_failures(100);

// set the maximum tries (50) and the soft uniqueness property
data_gen.anomaly_uniqueness().set_kind(kSoftUniqueness);
data_gen.anomaly_uniqueness().set_max_failures(50);
```

How to output a dataset?

One can output the dataset however she wants. The following function shows an example of how to output an anomalous step function dataset.

```
void example_step_anomaly()
{
  // generate the anomalous step function dataset
  data_list data = gen_step_anomaly();

  //--------------------------------------------------------------------------
  // print out the header information
  //--------------------------------------------------------------------------
  std::cout << "label" << "\t" << "time-stamp" << "\t";

  auto it =  data.begin()->variable_values().begin();

  for (; it != data.begin()->variable_values().end(); ++it)
  {
    std::cout << it->first << "\t";
  }
  std::cout << std::endl;

  //--------------------------------------------------------------------------
  // print out the actual generated data
  //--------------------------------------------------------------------------
  data_list::iterator i = data.begin();

  for (; i != data.end(); ++i)
  {
    std::cout << std::setprecision(4);
    std::cout << i->label() << ",\t";
    std::cout << std::setprecision(0);
    std::cout << i->time_stamp();

    std::map<std::string, double>::iterator it = i->variable_values().begin();

    std::cout << std::setprecision(4);
    for (; it != i->variable_values().end(); ++it)
    {
      std::cout << ",\t";
      std::cout << std::fixed << it->second;
    }
    std::cout << std::setprecision(0);

    std:: cout << std::endl;
  }
}
```

Makefile

A Makefile serves to compile the DATSA source code and any new dataset example.

You can find a sample Makefile in the datsa/examples folder.

Compile and Run

In the datsa/examples folder, one can build the DATSA executable (called “datsa”) and then execute it using the following command line instructions:

```
$ make clean
$ make
$ ./datsa
```

A Wrap-up Example

To conclude this tutorial. An anomalous cosine wave data generator is presented here for your reference:

1. The dataset has 10000 timestamps from 1 to 10000.
2. The normal duration is in the range of [10, 20), and the anomalous duration is in the range of [1, 100).
3. The anomalous event's probability is 1/50.

The cosine wave function with amplitude 1 and frequency 1 is defined as follows:

$$y = cos(\frac{2\pi}{360}i), \text{ where } i = timestamp \ \% \ 360$$

The anomalous function interrupts the cosine wave by adding random noise to an original cosine value at the corresponding timestamp.

Create an Anomalous Cosine Wave Example

1. Enter the ***datsa/examples*** folder
2. Create a folder named ***cosine***
3. Enter the ***cosine*** folder
4. Create the files ***cosine.h***, ***cosine.cpp***

cosine.h

```
#ifndef COSINE_H_
#define COSINE_H_

void example_cosine();
```

```

#endif // COSINE_H_
```

cosine.cpp

```
//-----------------------------------------------------------------------------
// include neccessary headers
//-----------------------------------------------------------------------------
#include "../../src/data_generator.h"
#include "../../src/function_variable.h"
#include "../../src/event_data.h"
#include "../../src/multithreaded_scheduler.h"
#include "cosine.h"
#include <iomanip>
#include <cmath>

#define AMP      1.0
#define FREQ     1.0

using namespace anomaly;

//-----------------------------------------------------------------------------
// define the normal value generation function
//-----------------------------------------------------------------------------
static double cosine_normal_function(event_data const &event)
{
  int i = (event.timestamp() - 1) % 360;
  double interval = (2*M_PI)/360;
  return double(AMP * cos(interval * i));
}

//-----------------------------------------------------------------------------
// define the anomalous value generation function
//-----------------------------------------------------------------------------
static double cosine_anomaly_function(event_data const &event)
{
  int MULT = 1;

  if (rand() % 2) MULT = -MULT;

  double LO = 0.2 * MULT;
  double HI = 0.5 *  MULT;
  double perturb = (static_cast <double> (rand()) /
    static_cast <double> (RAND_MAX)) * (HI-LO) +  LO;
  int i = (event.timestamp() - 1) % 360;
  double interval = (2*M_PI)/360;
  return double(AMP * cos(interval * i) + perturb);
}

//-----------------------------------------------------------------------------
```

```
// create the cosine anomaly dataset
//----------------------------------------------------------------------------
static data_list gen_cosine_function()
{
  // time duration
  int time_start = 1, time_end = 10000;

  // anomalous and normal duration range
  int normal_lowerbound_duration = 10;
  int normal_upperbound_duration = 20;

  int anomaly_lowerbound_duration = 1;
  int anomaly_upperbound_duration =  100;

  // anomaly probability nominator and deniminator
  long prob_num = 2;
  long prob_den = 100;

  // data generator declaration
  data_generator data_gen(time_start, time_end, normal_lowerbound_duration,
    normal_upperbound_duration, anomaly_lowerbound_duration,
    anomaly_upperbound_duration, prob_num, prob_den);

  // function variable for value generation
  function_variable *v = new function_variable("cosine-wave");
  v->normal_function(&cosine_normal_function);
  v->anomaly_function(&cosine_anomaly_function);
  data_gen.insert_variable(v);

  // parallel data generation scheduler as a back-end
  data_generation_scheduler *s = new multithreaded_scheduler(data_gen);
  data_gen.set_scheduler(s);

  // generate the data
  data_gen.generate_data();

  // return the dataset as a data_list object
  return data_gen.generated_data();
}

//----------------------------------------------------------------------------
// display the cosine anomaly dataset
//----------------------------------------------------------------------------
void example_cosine()
{
  data_list data = gen_cosine_function();

  //--------------------------------------------------------------------------
  // print out the header information
  //--------------------------------------------------------------------------
  std::cout << "label" << "\t" << "time-stamp" << "\t";
  std::map<std::string, double>::iterator it = data.begin()->variable_values().begin();

  for (; it != data.begin()->variable_values().end(); ++it)
  {
    std::cout << it->first << "\t";
  }
  std::cout << std::endl;

  //-------------------------------------------------------------------------
  // print out the actual generated data
  //-------------------------------------------------------------------------
  data_list::iterator i = data.begin();

  for (; i != data.end(); ++i)
  {
    std::cout << i->label() << ",\t";
    std::cout << i->time_stamp();

    std::map<std::string, double>::iterator it = i->variable_values().begin();

    std::cout << std::setprecision(4);
    for (; it != i->variable_values().end(); ++it)
    {
      std::cout << ",\t";
      std::cout << std::fixed << it->second;
    }
    std::cout << std::setprecision(0);

    std:: cout << std::endl;
  }

}
```

main.cpp

Create a main.cpp file in the datsa/examples folder if the main file does not exist:

```
//---------------------------------------------------------------------------
//---------------------------------------------------------------------------
#include <iomanip>
#include "cosine/cosine.h"

//---------------------------------------------------------------------------
//---------------------------------------------------------------------------
int main()
{
  srand(time(NULL));

  example_cosine();
  return 0;
}
```

If the main file already exists in the datsa/examples folder, include the header cosine/cosine.h, comment out other examples, and call the function `example_cosine()` like the following:

```
...
#include "cosine/cosine.h"

int main()
{
  ...
  //example_other();
  example_cosine();
  ...
}
```

The Makefile

If the Makefile does not already exist in the datsa/examples folder, one possible Makefile is provided here:

```
# for C++, replace CC (c compiler) with CXX (c++ compiler) which is used
# as default linker
CC=$(CXX)
CXXFLAGS=-fPIC -Wall -std=c++11 -O2 -g
LDFLAGS=-shared

LIBNAME=anomalydatagen
EXEC=datsa

LIBOBJS = ../src/data_generator.o ../src/event_data.o

# ===========================================================================
# Modify: append the object file to the OBJS variable
OBJS = main.o cosine/cosine.o
# ===========================================================================

# "all" is the name of the default target, running "make" without params would use it
all: lib$(LIBNAME).so $(EXEC)

lib$(LIBNAME).so: $(LIBOBJS)
    $(CC) ${LDFLAGS} -o $@ $^

install: lib$(LIBNAME).so
    cp lib$(LIBNAME).so /usr/local/lib
    ldconfig -v -n .

uninstall:
    rm -f /usr/local/lib/lib$(LIBNAME).so

$(EXEC): $(OBJS)
    $(CC) $(CFLAGS) -o $@ $^ -L. -l$(LIBNAME)

clean:
    rm *.so $(EXEC); rm ../src/*.o; find . -name "*.o" -type f  -delete
```

Otherwise, append the object file cosine/cosine.o to the OBJS variable of the existing Makefile like the following:

```
...
OBJS = main.o example1/example1.o example2/example2.o ... cosine/cosine.o
...
```

Run and Log the Anomalous Cosine Dataset

```
$ make clean
$ make
$ script cosine.log
$ ./datsa
$ exit
```

Example Scheduler Back-end

In this section, we will create a simple scheduler named `simple_scheduler` and later deploy it. The scheduler extends the `data_generation_scheduler` class and implements the `execute` function.

data_generation_scheduler

This is the parent class for schedulers.

The `data_generation_scheduler` class:

```
class data_generation_scheduler
{
public:

  data_generation_scheduler(data_generator &dg) : dg_(dg) {}

  virtual ~data_generation_scheduler() {}

  void virtual execute() = 0;
protected:

  data_generator &dg_;
};
```

simple_scheduler

The `simple_scheduler` class implements a single-threaded scheduler. To create such an example, you may follow through the following steps:

1. Derive `simple_scheduler` from the `data_generation_scheduler`
2. Declare the `simple_scheduler` constructor
3. Implement the `execute` function as follows
    - Determine whether an event is anomalous or normal
    - Determine the duration of the event
    - Generate data entries for the current duration
    - Iterate until the total time duration is reached

The `simple_scheduler` class:

```
class simple_scheduler : public data_generation_scheduler
{
public:

  //--------------------------------------------------------------------------
  // the constructor for the simple scheduler
  //--------------------------------------------------------------------------
  simple_scheduler(data_generator &dg) : data_generation_scheduler(dg) {}

  //--------------------------------------------------------------------------
  // the execute function generates and arranges data entries
  // dg_ is the data generator's reference declared in the parent  class
  //--------------------------------------------------------------------------
  virtual void execute()
  {
    // iterate until the total time duration is reached
    for (int timestamp = 0; timestamp <  dg_.time_end();)

      // determine whether an event is anomalous or normal
      anomaly::e_data_types event_type =
        dg_.is_anomaly_based_on_probability() ?
        kAnomalyDataType : kNormalDataType;

      // determine the duration of the event
      int duration = get_duration(event_type, timestamp);
      // generate data entries for the current duration
      dg_.generate_data_entry(timestamp, duration, event_type,
        dg_.variables(), dg_.generated_data(), dg_.time_step());

      // update the current timestamp
      timestamp += duration;
  }
private:

  //--------------------------------------------------------------------------
  // generates the duration based on the event type.
  //
  // if respect_windows() returns true, determine the normal or  anomalous
  // duration based on the event type and the current timestamp. Truncate  the
  // duration if it exceeds the end time.
  //
  // if respect_windows() returns false, return the duration directly.
  //--------------------------------------------------------------------------
  int get_duration(anomaly::e_data_types event_type, int  timestamp)
  {
    int duration =  dg_.generate_duration_based_on_type(event_type);

    if(dg_.respect_windows())
    {
        return duration+timestamp >  dg_.time_end()
          ? dg_.time_end() - timestamp :  duration;
     }
     else
     {
        return duration;
     }
  }

};
```

Deploy simple_scheduler

```
  // data generator defined elsewhere
  data_generator data_gen(...);
  ...

  // deploy simple_scheduler as the scheduler back-end
  data_generation_scheduler *s = new simple_scheduler(data_gen);
  data_gen.set_scheduler(s);
```