

# **Tutorial for PARK: Service, Script and GUI**

**Version 1.0.0**

***Wenwu CHEN***

## 1 PARK Service and Clients

### 1.1 Introduction

### 1.2 PARK service

### 1.3 Echo service

### 1.4 Echo client

### 1.5 Tips for service

## 2. Fitting service and client applications

### 2.1 Introduction

### 2.2 Fitting service

### 2.3 Fitting script client

## 3 Gaussian Model

### 3.1 Introduction

### 3.2 Define Parameter

### 3.3 Define Theory

### 3.4 Define Dataset

### 3.5 Define Data

### 3.6 Define MetaData

### 3.7 Script client applications

## 4. GUI framework for fitting client

### 4.1 Fitting GUI client

### 4.2 GUI client applications

#### 4.2.1 Model page

#### 4.2.2 Model Builder

#### 4.2.3 Dataset Representation

#### 4.2.4 Data Editor

### 4.3 Plug GUI components

## Summary

# **1 PARK service and client**

## **1.1 Introduction**

PARK is a simple and pluggable framework for distributed computing. It is implemented in pure python, provides the management functions for the job, service message queue and load balancing in distributed environment for computing. The PARK release provides:

- A pluggable framework for distributed computing so the user-defined services can be plugged into this distributed framework
- A pluggable framework for distributed fitting service so the used-defined models can be plugged into this distributed fitting framework.
- A pluggable framework for GUI client applications so the used-defined GUI components for models can be plugged into this GUI application framework, and be used to carry out distributed fitting service.
- A pluggable framework for script client applications so the used-defined script adapters for models can be plugged into this script application framework, and be used to carry out distributed fitting service.
- The local and global optimization, uncertainty estimation, and constrains and simultaneously multiple data and multiple models fitting.

The architecture of PARK is shown as in Fig. 1, and the working flow under PARK is shown as in Fig. 2.

The working flow under PARK is similar to the ordinary map-reduce pattern, which works as the following:

1. The client sent the service request to PARK server.
2. PARK server schedules the requested jobs, and assigns the job to the idle working node (worker) to perform the work.
3. Once the work is done by the worker, the results from the worker are archived in the server and applied by the reduce function, the final results are placed in message queue. At the same time, new service requests may be submitted by the reduce function (3.5 in Fig. 1).
4. The final results in message queue are submitted to the client applications.

PARK release provides the following functions:

1. PARK server/worker: provides a networking environment for the distributed service. The service can be plugged into PARK worker framework, and accessed by the client, while PARK server provides some basic management functions for the distributed services, such as job scheduling, load balancing, robustness.
2. Echo and Fitting service: They are two implemented services plugged into PARK server framework. Echo service can be used to testing the networking communication between the client and the server, and the server and the worker. Fitting service is a generic framework for data fitting and simulation. The third party can plug their models into the fitting service, and perform the data fitting

- and uncertainty estimation using the provided local and global optimization algorithms, such as the simplex algorithm, generic algorithm, etc, and parameter analysis algorithms.
3. Tools for script-based client applications: They provide the auxiliary classes and functions to handle the network communication and protocol, simplify the development of the script file-based client applications. They also provide the framework of the client application for the fitting service.
  4. GUI fitting client application: This provides a pluggable framework for the GUI fitting client applications. The user defined GUI components for the corresponding models can be plugged into this GUI framework.
  5. Other tools: PARK also provides many auxiliary classes and functions to help the developing of services and models, such as the classes to perform the transformation between xml string and the object tree, commonly used GUI components, etc.

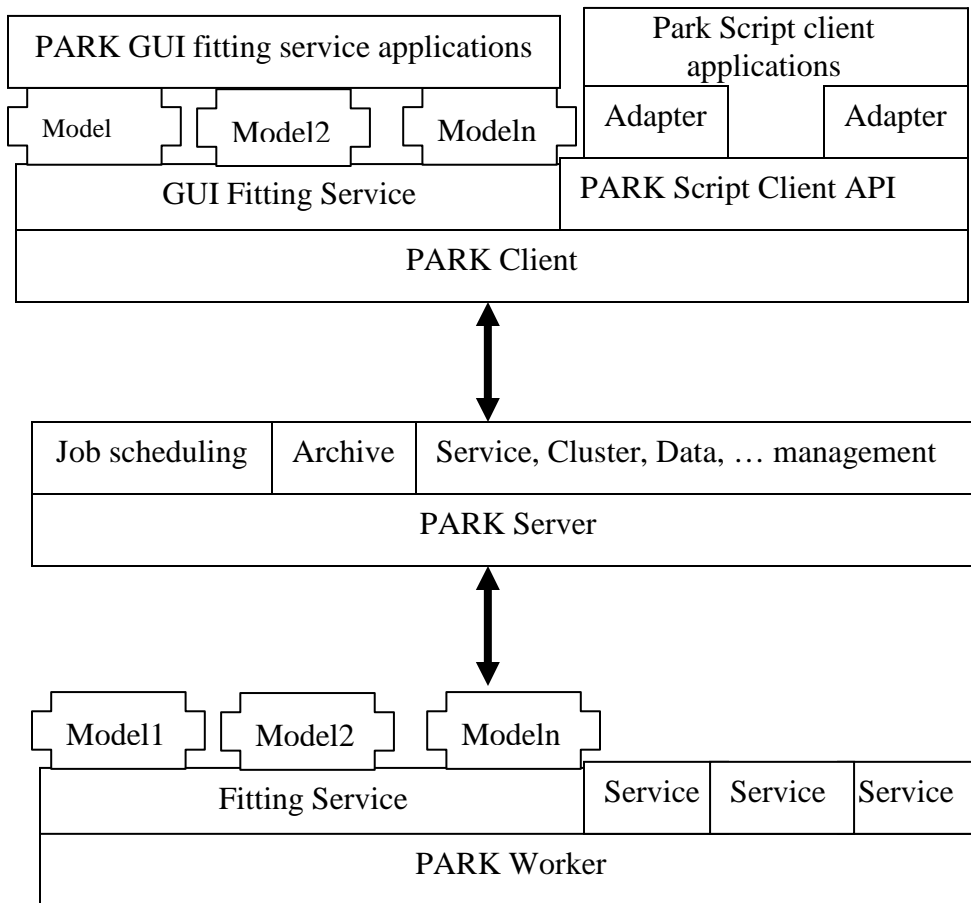


Fig. 1.1 The architecture of PARK

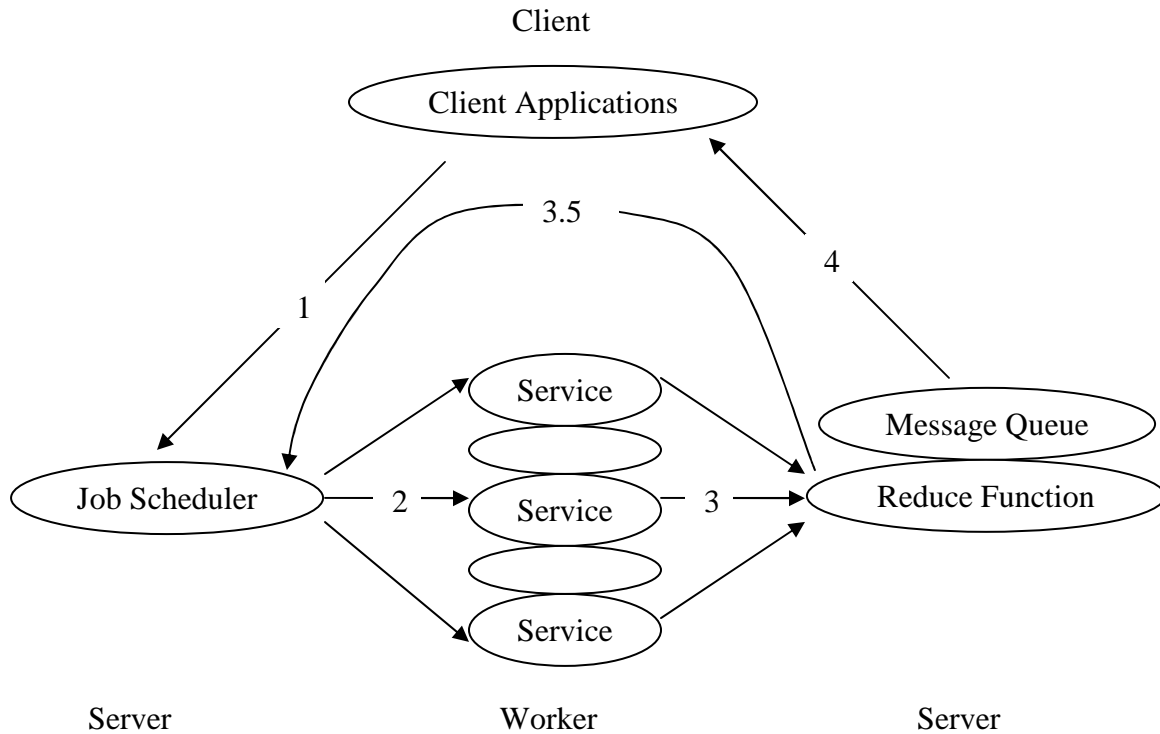


Fig. 1.2 The working flow under PARK

## 1.2 Park Service

For most developers and users, there are two kinds of programs running under PARK framework: the client applications that send service requests to PARK server, and the service that will be called by the PARK workers to do the real work.

A service under PARK framework is an executable program (it will be running in a separated process managed by PARK worker) or a python function (it will be running as a separated thread managed by PARK worker). The conventions for PARK services are:

1. For the executable program used as the PARK service, normally it will read input from the standard input, and send results to the standard output. The inputs from the standard input are the user's inputs sent by the client applications, and the outputs to the standard output will be collected by PARK worker, and sent back to the client applications. The client applications can also specify the files under local file system that the service can access as the input and output, however this is rarely used, for the service will run in distributed environments.
2. For the python function used as the PARK service, it should have the following declaring format:

```
void parkFunction ( input : string, sn = 1: int )
```

where input is a valid xml string sent by the user from the client application, sn is the serial number of a sequence function call. Within the function, everything sent to the standard output will be collected by PARK worker and sent back to the client applications

3. All the information is transferred as the xml string, so the input sent by the user and the output from the service must be, or can be transferred to the valid xml string. Furthermore, the input sent by the client application must be within the tags '<input>' and '</input>', which indicating the input string for the service.
4. Put the executable program or python function under the sites that PARK worker can access. The default site is  $\$PARK/services/$ .

For PARK is implemented in pure python, it use the multi-threading environment of python for multi-threading program. Python's multi-threading environment doesn't have good performance for multi-core computers, for it will use only one CPU no matter how many threads are running simultaneously. Multi-process running mode for PARK service is recommended for most case, especially when the service needs some time to be completed. It is also more robust that the multi-threading mode, it may need more startup time, and more resource.

### 1.2.1 Echo service

Let's see a simple service example, which just echoes the user inputs. The source code for echo service is in *service/commonsvr/echo.py*.

```
1. #!/usr/bin/env python
2.
3. #####
4. """ The Echo service that sends back the user's xml request """
5. from commonsvrUtil import service, serviceFile
6. #####
7.
8. #####
9. def echo(s0, sn = 0):
10.     print s0
11. #####
12. #####
13. FILENAME = os.path.join(os.getcwd(), 'echo.xml')
14. ECHO_SERVICE = 'Echo service'
15. #####
16.
17. #####
18. if __name__ == '__main__':
19.     #serviceFile(echo, FILENAME, ECHO_SERVICE)
20.     service(echo, ECHO_SERVICE)
```

---

---

The echo service can be used as a function or an executable program. The service name is 'echo.py' for the executable program or 'echo' for python program. The format of function echo agrees with rule 2 in section 1.2 (line 9). This function is very simple, it just prints out the user input to the standard output, as shown in line 10, which will be collected by PARK worker and sent back to the user.

This program also use the utility functions defined in commonsvrUtil.py: service, serviceFile, whcih can be used to simplify the developing of python services. They just read the inputs from the standard input or from a file, and then call the specified function. Their source code is shown in the following:

```
#####
""" read from stdin and return the string for service. """
## A special tag to separate the serial number and the service request
SPECIAL_COUNT_TAG = '##'

## length of the special tag
LEN_SPECIAL_COUNT = len(SPECIAL_COUNT_TAG)

#####
def getString(fd):
    """ Read the service request string from stdin. """
    s0 = fd.read()
    strname = s0.strip()
    ind = strname.find(SPECIAL_COUNT_TAG)

    sn = 0
    if ind >= 0:
        try:
            sn = int(strname[0:ind])
        except:
            pass

    ind += LEN_SPECIAL_COUNT
    return (s0[ind:], sn)

    else:
        return (s0, sn)
#####

def service(func, serviceName=None):
    """ The service framework.
        func is the service function, which has the form:
        func(sn, s0) where sn is the serial number, and
        s0 is the service request information.
    """
    try:
        (s0, sn) = getString(sys.stdin)
        func(s0, sn)
    except:
        if serviceName is None:
            print 'Service exception:%s'\
                %(traceback.format_exc())
        else:
```

```

        print 'Service %s exception:%s'\
              %(serviceName, traceback.format_exc())
#####

#####
def serviceFile(func, fname, serviceName=None):
    """ The service framework.
        func is the service function, which has the form:
        func(sn, s0) where sn is the serial number, and
        s0 is the service request information.
    """
    fd = None
    try:
        fd = open(fname)
        (s0, sn) = getString(fd)
        fd.close()
        return func(s0, sn)
    except:
        if fd is not None:
            fd.close()

        if serviceName is None:
            print 'Service exception:%s'\
                  %(traceback.format_exc())
        else:
            print 'Service %s exception:%s'\
                  %(serviceName, traceback.format_exc())

#####

```

service is used for the PARK service to run as a executable program, while serviceFile is mainly used for the local debugging. Always make sure that the service works under local machine with the similar environments of the distributed machines.

### 1.3 Park client

The developing of PARK service is simple, because the network communication between the PARK server and PARK worker, the work-flow, and other features, such as job management, process and thread management, archiving, etc, has been defined in PARK. This is a little different from the PARK client, where the user provides the inputs for the service and analysis the output from the service. That working flow total depends on the service. Although PARK cannot provide all the functions for all the service, it provides some common functions for the developing of PARK client, especially the functions to handle the network communications. Based on these tools, the developer can develop some more user-friendly tools specified for the specific service.

The UML diagram for PARK client API is shown in Fig. 2. The implementation is under \$PARK/script/. In this section, the common functions are described and some examples of the echo client applications are explained. The specified tools for

Gaussian fitting using the fitting service will be investigated in more details in the following chapter.

Fig. 1.3. UML diagram for PARK client API.

### 1.3.1 A simple echo client

The most important class of PARK client API is NetworkThread class, which can run as a separate thread to handle the network communication with PARK server by a NetworkClient object. The thread using the NetworkThread object must provide a NetworkHandler object to support the queuing storage of the service request, and handle the reply from the server, and other features, such as the handling of the error during the network communication. These classes are not limited only for the communication of PARK, they are generic enough and can be also applicable for the networking communication in python environments. The following are some important functions for the individual class:

```
class NetworkThread:
    def __init__(self, server, handler, timeout=1.0)
        The constructor of NetworkThread class. Its parameters are:
        server: a list of ( serverName:string, port:int) to specify the
                remote server
        handler: an object of NetworkHandler classes to handle the
                request to the server, and the reply from the server.
        timeout: a float number to indicate the timeout time for the
                polling

    def start(self):
        Start the network thread to handle the network communication to
        the server

    def stop(self):
        Stop the network communication with the server

    def is_running():
        Return True if the thread is running to handle the network
        communication

class NetworkHandler:
    def __init__(self):
        The constructor.

    def GetOutQueue(self):
        Return a queue so the client can submit the request into it.

    def ProcessMessage(self, mtype, msg):
        Process the message related to the network communication. The
        parameters are:
        mtype: integer, the type of network message.
                Currently, the following types are defined:
                NETWORK_CONNECT = 0: connect to the server
                NETWORK_CLOSE = 1 : close the connection to the server
                NETWORK_REPLY = 2 : reply from the server
```

```

        NETWORK_ERROR = 3 : error or exception from the network
msg: string, the related network message.

def _OnMessage(self, msg):
    Handler for the normal reply from the server. Called by
ProcessMessage for mtype == NETWORK_REPLY

def _OnConnect(self, msg):
    Handler for the connection to the server. Called by
ProcessMessage for mtype == NETWORK_CONNECT

def _OnClose(self, msg):
    Handler for the connection to the server is closed. Called by
ProcessMessage for mtype == NETWORK_CLOSE

def _OnError(self, msg):
    Handler for the error or exception during the network
communication. Called by ProcessMessage for mtype == NETWORK_ERROR.

```

Based on these classes, we can implement the client examples to communicate with PARK server. The example of the client application to use the echo service is shown in \$PARK/script/echoHandler.py and \$PARK/script/echoClient.py.

In order to do that, we first need to implement the network handler to handle the message from the server. The following is the implementation of EchoHandler class, which just prints the reply in the function \_OnMessage():

```

#!/usr/bin/env python

#####
"""
    The class to handle the reply from the park server
"""
#####

from networkHandler import NetworkHandler
#####

#####
class EchoHandler(NetworkHandler):
    """
        A network client class to send/receive network
data asynchronously. The output data are stored
in outqueue, and the input data are handled by
inhandler. The items in Queue must be castable
to a string. This is the class used by the
network thread. Not used directly by the user.
    """
#####
    def __init__(self):
        super(EchoHandler, self).__init__()
#####

```

```

def _OnMessage(self, msg):
    """
        Handler for the normal reply from the PARK server.
        The subclass must implement this function.
    """
    print 'Reply from parkServer:', msg

```

```

#####
#####

```

Then we can have a client application that uses the implemented network handler:

```

#!/usr/bin/env python

#####
import traceback
#####

from echoHandler import EchoHandler
from networkThread import NetworkThread

#####
## default PARK servers:
DEFAULT_SERVERS = [socket.gethostname(),
                   'localhost', 'compufans.ncnr.nist.gov']
## default port number
DEFAULT_PORT = 5400

## DEFAULT WAITING TIME
WAITING_TIME = 150
#####

#####
def main():
    server = (DEFAULT_SERVERS[0], DEFAULT_PORT)
    handler = EchoHandler()

    networkThread = None

    try:
        networkThread = NetworkThread(server, handler )
        networkThread.start()
        print '\n --- Start network Thread ----'

        request = """
            <session version='0.2.1' name='wwchen' type='7' user='wwchen'
                email='wwchen@nist.gov' priority='0'>
            <group name='group1'><dataSet></dataSet>
            <reduce classname='Chisq'/>
            <task cmd='echo.py'><bufsize value='3000'/></task>
            <joblist name='joblist1' cnt='2'>
                <input>This is a echo example</input></joblist>
            </group></session>
            """

```

```

handler.GetOutQueue().put(request)

print '\n --- Main Thread is sleeping ----'
time.sleep(WAITING_TIME)

except:
    print 'NetworkThread exception: ', traceback.format_exc()

if networkThread is not None:
    print '\n --- Stop network Thread ----'
    networkThread.stop()

print '\n --- Main Thread is closing ----'
#####

if __name__ == '__main__':
    main()
#####

```

The request is specified by request string in the client application. For the request, besides the user input which is labeled with '<input>' and '</input>' and will be sent to the service program as the input of the program, there are some other tags which will be handled by PARK server. Some important ones are the attribute of cmd='echo.py' with <task> tag which define the remote service name to be used, and the attribute of cnt='2' within <joblist> tag which specify the number of service to be run. The sn parameter for the service function is in the range of [0 .. cnt].

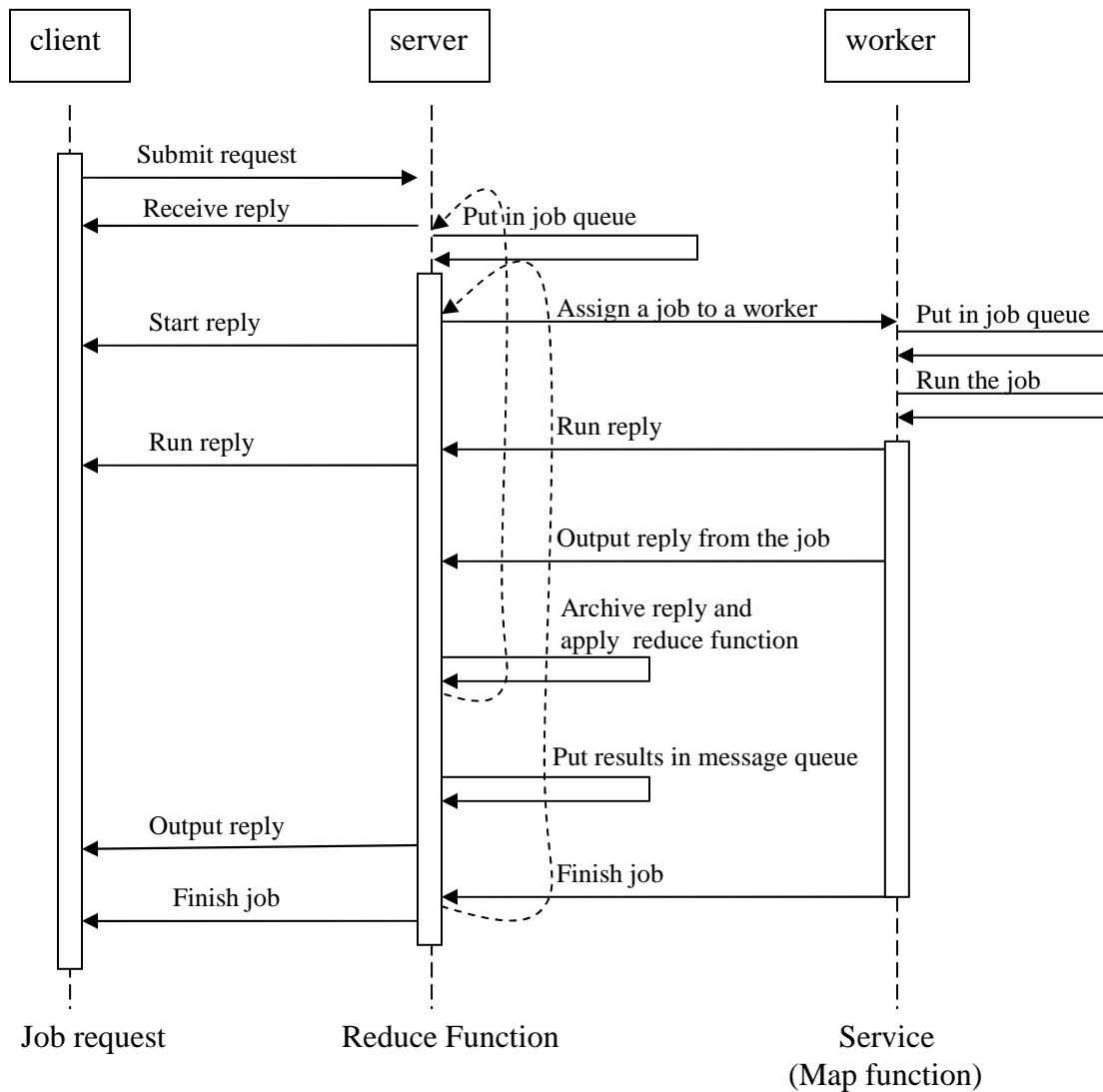
When you run this client application successfully, the EchoHandler object will print out the reply from the PARK server each time when these is the reply from the server. Remember to start the PARK server first.

### 1.3.2 High-level of client API

These are the low-level of PARK client API, they are enough to develop the client applications once we know the format of xml request and reply. However, these low-level client API is rarely used for client application, for they involve the details information about the service request, these are nor user-friendly and occasionally changed. The developer of the client applications wants to focus on the handling of user input, not the format about the service request and service reply. Specified for the communication under PARK framework, we can refine the client API, and provide more specific and more friend client API for the client application developers.

Before the explaining of client API for PARK, let's see the communication under PARK framework first, which is illustrated in Fig. 3.

Fig. 1.3 The UML sequence diagram for PARK framework



The network handler for the communication of PARK is ParkHandler. It inherits from NetworkHandler. Besides overriding the functions defined by NetworkHandler, it further refines the function of `_OnMessage()` functions to handle reply for the receiving, starting, running, output, and finishing of the job.

Beside the service request sent to PARK server for the service, PARK server also support some other features, such as stop the job, query the job queue status, register as a listener for the specified job, etc. For these requests are handled by PARK server directly, and are not associated to a specific job, ParkHandler defines the interface to handle the reply for these replies. More details about ParkHandler can be found in its API document.

ParkHandler provides functions to handle the underneath network communication, while ParkScriptClient provides more user-friendly interface for the developer of client applications. ParkScriptClient defines some commonly used interfaces, and it can be further refined by the client to use the specific service, for example, the ParkEchoClient is a specialized client interface to use echo service.

Some important functions for ParkScriptClient class are:

```
#def _getDefaultSessionTemplate(self):
# Return a ParkServiceSession object, from which the user can
change some #properties of the service request, such as the user
name, project name, priority, the #service name, etc.

def submit(self, job, count=1):
    submit a job as the input for the service, and run it count
times. Job must be a valid xml string, or the object that can be
transferred to a valid xml string by the function obj.toStream(),
obj.toXml() or str(obj). The xml string must start with '<input>'
and end with '</input>' tag. That xml will be used as the input of
the service.

def isCompleted(self):
    Return True if all the requested jobs have been done.

def getJob(self, block=True):
    Return a finished job. If block = True, it will be blocked until
there is a job available, otherwise, it will return immediately even
there is not finished job available.
```

The parkClientUtil.py provides two functions to simplify the client applications: runParallel() and runSequence(), their source code is shown in the following:

```
def runParallel(client, job_counts, jobHandler ):
    """
    A list of jobs can be run parallel.
    Client: an object of ParkScriptClient
    job_counts: a list of (job, count) pair
    jobHandler: a handler for the finished job.
    """
    try:
        for item in job_counts:
            client.submit(item[0], item[1])

        while (not client.isCompleted()):
            job = client.getJob()
            jobHandler(job)
    except:
        print ' exception:', traceback.format_exc()

    if client is not None:
        client.stop()
#####

def runSequence(client, job, counts, jobHandler ):
```

```

"""
    A list of jobs that should be running sequentially.
    Client: an object of ParkScriptClient
    job: the original job
    counts: a list of counts for each job
    jobHandler: a handler for the finished job. It should return
               a new job for the next request.
"""
try:
    myjob = job; ind = 1
    for cnt in counts:
        client.submit(myjob, cnt)
        newjob = client.getJob(True)
        myjob = jobHandler(newjob, job, cnt, ind)
        ind += 1

except:
    print 'exception:', traceback.format_exc()

if client is not None:
    client.stop()
#####

```

runParallel() runs a list of jobs in parallel, all the jobs are submitted to the server simultaneously, and the jobhandler function will be applied for each finished job. runSequence() will run a list of job in sequence, the jobHandler function will be applied for the finished job, and return a new job to be submitted again.

### 1.3.4 Echo Client applications using high-level client API

```

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

import time, sys
import getopt
import traceback
#####

from parkScriptClient import ParkScriptClient
from parkEchoHandler import ParkEchoHandler
from parkScriptClient import DEFAULT_SERVER, DEFAULT_PORT
#####

## echo service
ECHO_SERVICE = 'echo.py'
#####

#####

class ParkEchoClient(ParkScriptClient):
    """ The script client for echo service.
        It may be used to testing the communication between
        the client and server, and the workers, for all
        the sent-out messages and received messages can be print out.
    """

```

```

"""
#####

def __init__(self, server=DEFAULT_SERVER,
             port = DEFAULT_PORT):
    """ Constructor. """
    ParkScriptClient.__init__(self, server, port)
#####

def _getDefaultHandler(self):
    """ Return the handler for the network. """
    return ParkEchoHandler()
#####

def _getDefaultSessionTemplate(self):
    """ Return the default template for the service request """
    session = ParkScriptClient._getDefaultSessionTemplate(self)
    session.setService(ECHO_SERVICE)
    return session
#####

#####

def setEchoFlag(self, echoFlag=True):
    """
        Set flag to decide whether echo the reply or not.
        Mainly used for the testing.
    """
    self.getHandler().setEchoFlag(echoFlag)
#####

def setSubmitFlag(self, submitFlag=True):
    """
        Set flag to decide whether print the request or not.
        Mainly used for the testing.
    """
    self.getHandler().setSubmitFlag(submitFlag)
#####

#####
# The following are some testing patterns for echo service.
#####

def printEchoJobsInfo(echo):
    """ Print the information of the jobs handled by the client. """
    print 'Is job finished:', echo.isCompleted(), \
          ' Number of total jobs:', echo.getJobsCount(), \
          '\nHas Waiting jobs:', echo.hasWaitingJobs(), \
          ' Number of Waiting jobs:', echo.getWaitingJobsCount(), \
          '\nHas Running jobs:', echo.hasRunningJobs(), \
          ' Number of Running jobs:', echo.getRunningJobsCount(), \
          '\nHas Finished jobs:', echo.hasFinishedJobs(), \
          ' Number of Finished jobs:', echo.getFinishedJobsCount()
#####

#####
WAITING_TIME = 60
COUNT = 2

```

```

SERVICE_REQ_NUM = 2
#####

#####
def test0():
    """ Testing example that the main thread sleeps long enough to
        waiting for the completing of the service request.
    """
    echo = None
    try:
        echo = ParkEchoClient()

        req = '<input> This is a echo client </input>'
        echo.submit(req, COUNT)

        print '\n Echo Jobs Info before sleeping:'
        printEchoJobsInfo(echo)

        time.sleep(WAITING_TIME)

        print ' Echo Jobs Info after sleeping:'
        printEchoJobsInfo(echo)

        job = echo.getJob()
        print '\n finished job:', job

        print '***** finish testing *****'

    except:
        #pass
        print ' exception:', traceback.format_exc()

    if echo is not None:
        del echo

#####
def test1():
    """ Testing example that the main thread continues to polling
        whether the service requests have completed.
    """

    echo = None
    try:
        echo = ParkEchoClient()

        req = '<input> This is a echo client </input>'
        echo.submit(req, COUNT)

        print '\n Echo Jobs Info before sleeping:'
        printEchoJobsInfo(echo)

        while (not echo.isCompleted()):
            job = echo.getJob()
            print 'job finished:', job

        print '\n Echo Jobs Info after polling:'
        printEchoJobsInfo(echo)

```

```

        print '***** finish testing *****'

    except:
        #pass
        print ' exception:', traceback.format_exc()

    if echo is not None:
        del echo
#####

#####
def test2():
    """ Testing example that the main thread have more than
        one service request, it continues to polling
        whether the service requests have completed.
    """

    echo = None
    try:
        echo = ParkEchoClient()
        echo.setEchoFlag(False)

        for ind in xrange(SERVICE_REQ_NUM):
            req = '<input> This is a echo client: %i </input>' %(ind)
            echo.submit(req, COUNT)

            print '\n Echo Jobs Info at iter %i:' %(ind)
            printEchoJobsInfo(echo)

            while (not echo.isCompleted()):
                print '\n Echo Jobs Info while polling job'
                printEchoJobsInfo(echo)
                job = echo.getJob()
                print 'job finished:', job

            print '\n Echo Jobs Info after polling:'
            printEchoJobsInfo(echo)

            print '***** finish testing *****'

    except:
        #pass
        print ' exception:', traceback.format_exc()

    if echo is not None:
        del echo
#####

#####
def jobParallelHandler(job):
    """
        Job handler for parallel running mode.
    """
    print 'finished job:', job

```

```

#####

def jobSequenceHandler(newjob, job, cnt, ind):
    """
        Job handler for sequential running mode.
    """
    print 'finished job:', newjob
    print 'old job', job
    print 'index:', ind, ' count:', cnt

    return '<input> This is a echo client %i </input>' %(ind)
#####

#####

from parkClientUtil import runParallel, runSequence
#####

def test3():
    """ Testing for using runParallel function. """
    jobs = []

    for ind in xrange(SERVICE_REQ_NUM):
        req = '<input> This is a echo client: %i </input>' %(ind)
        jobs.append((req, COUNT))

    echo = ParkEchoClient()
    echo.setEchoFlag(False)

    runParallel(echo, jobs, jobParallelHandler)

    print '***** finish testing *****'

#####

def test4():
    """ Testing for using runSequence function. """
    counts = [COUNT for ind in range(SERVICE_REQ_NUM)]

    req = '<input> This is a echo client: 0 </input>'

    echo = ParkEchoClient()
    echo.setEchoFlag(False)

    runSequence(echo, req, counts, jobSequenceHandler)

    print '***** finish testing *****'

#####

def usage():
    """ The usage of this testing """
    s0 = 'Usage: parkEchoClient [-h] [--help] [--t testing_case] ]\n'
    s0 += 'Options:\n'
    s0 += '    -h          Display this information\n'
    s0 += '    --help    Display this information\n'
    s0 += '    --t case   The case for the testing\n'
    s0 += '    case has the following meaning\n'

```

```

    s0 += '    1    testing parkScriptClient.isCompleted() for a
single request.\n'
    s0 += '    2    testing parkScriptClient.isCompleted() for
multiple requests.\n'
    s0 += '    3    testing parkClientUtil.runParallel() for
multiple requests.\n'
    s0 += '    4    testing parkClientUtil.runSequence() for
multiple requests.\n'
    s0 += ' others testing parkScriptClient.getJob() after long
sleeping.'
    print s0

#####
def main():

    testCase = TESTING_CASE

    try:
        opts, args = getopt.getopt(sys.argv[1:], 'ht', ['help', 't='])
    except getopt.GetoptError:
        usage()
        return 2

    for o, a in opts:
        #print o, a
        if o == ('-t', '--t'):
            try:
                testCase = int(a)
            except:
                pass

        if o in ('-h', '--help'):
            usage()
            return 0

    if testCase == 1:
        # self-controlled waiting for the completing of job.
        print 'Testing parkScriptClient.isCompleted() for single
request'
        test1()
    elif testCase == 2:
        # self-controlled waiting for the completing of multiple jobs.
        print 'Testing parkScriptClient.isCompleted() for multiple
requests'
        test2()
    elif testCase == 3:
        # using parkClientUtil.runParallel()
        print 'Testing parkClientUtil.runParallel()'
        test3()
    elif testCase == 4:
        # using parkClientUtil.runSequence()
        print 'Testing parkClientUtil.runSequence()'
        test4()
    else:
        # exampele using sleeping.
        print 'Testing parkScriptClient.getJob() after long sleeping
for single request'

```

```

        test0()

    return 0

#####
if __name__ == '__main__':

    sys.exit(main())

```

---



---

In this file,

- In function test0(), the main thread of the client application is given longer enough time to wait for the completing of the service, and then print the updated results.
- In function test1(), the main thread of the client application submits a job request which will run multiple times, then the main thread uses the getJob(block=True) to poll whether there is finished job. The isCompleted() is important, otherwise, the getJob(block=True) will be blocked for ever for there will not available finished job after the last finished job is pull out.
- In function test2(), it is similar to function test1() except that more than one job request have been submitted.
- In function test3(), the function runParallel() is used to submit a list of jobs.
- In function test4(), the function runSequence() is used to submit a list of jobs.

In these echo client applications, the inputs that the user sends to the service are just the valid xml strings, which can be assigned directly. There are no extra-steps to build the inputs for the service request. In real client applications, such as for the applications to use the fitting service, extra-steps may be needed to build the service input. These steps are service dependent, and it's better to provide some adapter classes to simplify the steps to build the inputs for the services, and also the handler to handle the reply from the service. Normally the following adapter classes may be implemented to simplify the development of PARK client applications:

- \*Handler class to handle the network communication. It may inherit from parkHandler.ParkHandler or networkHandler.networkHandler
- \*Client class to provide the interface for client applications. It may inherit from parkClient.ParkClient.
- \*ModelBuilder class to provide the functions to build the input for the service.

As an example, fitting service is implemented as a generic framework for the data fitting. PARK provide the following adapter classes to build inputs for fitting request:

- parkFit.ParkFit: generic interface and implementation to build fitting input
- parkUniFit.ParkUnit: generic interface and implement to build fitting that all the models are of the same type.
- parkGaussFit.ParkGaussFit: the adapter class to build Gaussian models for the fitting.

- `parkFitClient.ParkFitClient`: generic client interface for client applications using the fitting service
- `parkFitHandler.ParkFitHandler`: the underneath network handler for the reply from fitting service.

The implementation of these classes is closely related to the implementation of fitting service. They will be explained in more details in the next chapter.

## 2. Fitting service and its script client application

### 2.1 Introduction

The fitting service is implemented and plugged in PARK framework as a generic framework of data fitting and simulation, it provides functions of global optimization, uncertainty estimation, constrains of the parameters, etc. UML diagram for Fitting service classes Fig. 2.1.

Fig. 2.1 The UML diagram for fitting service.

The interface of fitting service is very simple, which is shown in services/commonsvr/fitting.py.

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

#####
# The generic fitting service.
#####

#####
from commonsvrUtil import setEnv
from commonsvrUtil import service, serviceFile

setEnv()

from xmlFitting import XmlFitting
#####

#####
""" The engineer for the fitting service"""
def fitting(s0, n0 = 1):

    fitting = XmlFitting()

    fitting.parseString(s0)

    print fitting.doFitting().toStream()

#####

#####
""" The fitting service gets the inputs from the file.
    This is a testing example that does the fitting for
    multiple Gaussian functions.
    """
#####
FITTING_SERVICE = 'Generic Fitting service'
#####

#####
if __name__ == '__main__':
    service(fitting, FITTING_SERVICE)
```

#####

Each fitting is described by an XmlFitting object, which contains one XmlMultiplexor and one XmlOptimizer object, and an interface *doFitting()* to perform the fitting and return the fitting results.

XmlOptimizer is a bridge between the XmlFitting object and the real optimizer implementation. The two most import functions are:

*void setOptClassName(string className)*  
*Optimizer getOptimizer()*

The first function set the class name of the optimizer that implements various optimization algorithms. The optimizer class needs to implement the interfaces defined in optimizer.parkOptimizer.Optimizer. The second returns the object of the specified optimizer.

The XmlMultiplexor includes the following components:

An collecton collection of XmlModel: each model provides the individual objective function and theoretical function, and has the associated dataset.

An XmlVariables object: the collection of XmlVariable who defines the variable definition, such as whether the variable is fixed, optimized, or get the value from an expression, set the range of optimized variable,

An XmlConstrains object: the collection of XmlConstrain who defines the constrains within or among the variables of the models.

The functions of XmlMultiplexor that are close related to the fitting are:

*double getObjectFx():* Return the objective function which is used for optimization  
*double getResidual(double[] p0, string modelName):*  
Return a function to calculate the residual of the model specified by the modelName, It is used to calculate the uncertainty in least-square problem.

The XmlVariable defines the variable definition for the fitting. It has a target property to indicate which variable is defined, a flag to indicate whether it is fixed, optimized, or a constrained by getting it value from a constrain expression. It also has a range to indicate the box range if it is the optimized variable. Currently the target of an XmlVariable is represented by three levels:

modelName.parameterName.attributeName. For example, it a model's name is 'M0', which has a parameter name 'pm0', and that parameter has attributes with names of ('a0', 'x0'), the target names for these two attributes are 'M0.pm0.a0' and 'M0.pm0.x0'. These target names can be used in the constrain expression for the constrained variables.

The XmlConstrain defines the constrain of an attribute with other variables by a constrained expression within the fitting. Currently this constrain only support the

simple constrains, which have only two parts: the target and the expression, and the constrain is equivalent to the expression of 'target = constrain\_expression'.

The XmlModel class is an important part for the fitting representing a logical model associated a set of data. It may, or may not correspond to a real physical mode. It is the wrapper of a set of close related data (experimental data or dummy data for the simulation) and a set of parameters to describe the model that simulates the data.

The XmlModel needs to provide the required functions for the fitting, such as the objective function for the optimization, and residual function for the uncertainty calculation in the least-square problem. Some important functions are:

***XmlMode(string & name):*** *Constructor. It should have a name, which the XmlMultiplexor uses to identify the model*

***XmlDataset getXmlDataset():*** *get the dataset*  
***void setXmlDataset(XmlDataset& dataset):*** *set the dataset*

***void add(XmlParameter & pm):*** *append a parameter to model*  
***void remove(string & pmName):*** *delete a parameter by its name*  
***void insertXmlParameter(int ind, XmlParameter & pm):***  
*Insert a parameter at position ind*  
***void clearXmlParameter():***  
*Remove all the parameters*  
***XmlParameter[] getXmlParameter():***  
*Return all the parameters*

***void setTheoryClassName(string theoryClassName):***  
*set the class name of the theory*  
***Theory & getTheory ():*** *Return the theory object*  
***void setXmlTheoryData (XmlTheoryData & data):*** *set the theory data object*  
***XmlTheoryData & getXmlTheoryData ():*** *get the theory data object*  
***void updateXmlTheoryData ():*** *update the theory data object*

The Theory class defines the simple interface for all theoretical subclasses will implement. At very low level, the following functions may be overridden (At least one function must be overridden):

***double \_getObjectiveFx (double \* data, XmlParameter [] pm):***  
*Return the objective function value, used for optimization*  
***double[] \_getResidual (double \* data, XmlParameter [] pm):***  
*Return the residual array, used to calculate the uncertainty in least-square problem*

At high level, the subclasses can override the following functions:

***double getObjectiveFx ():***

*Return the objective function value, used for optimization*

***double[] getResidual ():***

*Return the residual array, used to calculate the uncertainty in least-square problem*

with the help functions of:

***XmlDataset getDataset():*** *get the dataset*

***void setDataset(XmlDataset& dataset):*** *set the dataset*

***XmlParameter[] getParameters():*** *get the parameters*

***void setParameters(XmlParameter[] pms):*** *set the parameters*

The XmlDataset is a class to wrap all the information about the data, it includes a collection of XmlData objects, an XmlMetaData object, and an XmlReductionData object. This is an abstract class, its subclass need to implement the following pure virtual functions:

***void \_checkXmlReductionData (XmlReductionData & redData):***

*Correct the contents of the reduction data to match the results from theory*

***void \_updateXmlReduction ():***

*Update the reduction data when the meta data or data are changed.*

The important member of XmlDataset is XmlReductionData object, which is obtained from the collection of XmlData objects and the XmlMetaData object, and used by the theory class to perform the theoretical calculation.

The XmlData object is the individual data source for the dataset. It includes the data source and the associated meta data. XmlData class is also an abstract class, and the subclasses need to implement the following functions:

***void update4Source ():*** *update the contents due to the change of the source*

***void update4Meta ():*** *update the contents due to the change of the meta data*

If the data source is local, file-based source, the subclasses of XmlData may implement the following interfaces to separate the logical operations on the object and the read/write operations of the data source:

***void \_readRawData ():*** *read data from the source*

***void \_writeRawData ():*** *write data to the source*

If the data source is remote source, the subclasses of XmlData may implement the following interfaces to separate the logical operations on the object and the fetch/send operations between the local and remote file system:

*void \_fetchDataSource ():*    *fetch the data from the remote data source*  
*void \_sendDataSource ():*    *send and store the data at the remote data source*

The XmlData may or may not have the XmlReductionData or/and XmlMetaData object, they may be from the associated the source (for data fitting) or created manually (for the simulation).

The XmlTheoryData and XmlReductionData are just the collection of the data object (XmlData and its subclasses). The XmlData class is the wrapper of the data, its three subclasses have been defined:

XmlDataArray:        corresponding to 1D array in numpy  
XmlDataMatrix:      corresponding to 2D matrix in numpy  
XmlDataNArray:      corresponding to N-dimension array in numpy

The above mentioned classes inherit either from XmlAttribute or XmlObject (XmlObject also inherits from XmlAttribute). Both provide the basic operations, such as the automatic transferring between the xml string and the objects. In XmlAttribute class, all members are considered as the attributes in the corresponding xml DOM tree. In XmlObject class, all members that are added by call function **add(obj)** are considered as the child node in the corresponding xml DOM tree. Normally XmlAttribute is used when its members are simple types data, such as the built-in data type in python, while XmlObject is used when its members are objects.

The members of XmlAttribute class can be accessed directly by its name, for example:

```
attr = XmlAttribute('attr')
attr.data1 = 'Hello'
attr.data2 = 'Python'
print attr.data1, attr.data2,attr
# this will show "Hello"  "Python"
#      "<attr data1='Hello' data2='Python' />"
```

The members of XmlObject that is added by calling **add(obj)** class can also accessed directly by its name if *obj.name* member exists, for example:

```
obj = XmlObject('obj')
attr = XmlAttribute('attr')
attr.data1 = 'Hello'
attr.data2 = 'Python'
attr.name = 'a0'
obj.add(attr)

print obj, obj.a0
# this will show <obj><attr data1='Hello' data2='Python'
# name='a0' /> </obj>"
# and <attr data1='Hello' data2='Python' name='a0' />
```

The more details about the individual classes can be found in the manual of classes.

In order to plug a new model to this fitting service, the developer needs to implement the following steps:

- 1) Define the parameters to describe the model
- 2) Define the dataset, including the data format, and meta data for the model
- 3) Implement the theory class for the model
- 4) Optionally, the new optimization class may be implemented
- 5) Testing the model to make sure it works
- 6) Put the model within the place that the fitting service can access

In chapter 3, we will implement a simple model that performs the multiple Gaussian functions fitting as an example to explain how to plug a new model in this fitting service.

## 2.2 Fitting script client

The input for the fitting service is a valid xml string representing all the information required by the XmlFitting object to perform the fitting. Theoretically, the fitting input can be built directly by XmlFitting class to populate the required information, which provides all the functions to handle that. The GUI client application for fitting service is based on this mode to build the input for the user wants to build each components interactively. The condition is a little different for the developing of the script based client applications. The XmlFitting class is designed to do the data fitting, it provides many function related to the fitting, which may not be needed by the script client application. The client application would prefer a simple way to access the important information efficiently. The adapter classes can be developed to show only the more important functions to the client applications. However, these adapter classes only simplify the operations to build the input for service, and they are highly associated with the specified service.

PARK provides several adapter classes to simplify the procedure for building inputs for the fitting service:

- parkFit.ParkFit: a generic adapter class to build inputs for the fitting service
- parkUniFit.ParkUniFit: a generic adapter class to build inputs for the fitting service where all the models are of the same type.
- parkGaussFit.ParkGaussFit: the adapter class to build the inputs for the fitting service where all the models are of the Gaussian model type.

PARK also provides a utility class, ParkFitHandler, inheriting form ParkHandler, to handle the fitting reply from the fitting service. All the source code of these classes can be found at \$PARK/script.

Some important functions for ParkFit classes are:

```
def getOptimizer(self):  
    """ Return an XmlOptimizer object to set the control parameters  
    for optimization. The user can use this object to change the control  
    parameters for the optimizer.
```

```

    """

def setOptimizerClass(self, optClass):
    """ set the name of the optimization class to be used. """

def getVariables(self):
    """ Return a list of XmlVariable object, where the user can use
    it to define whether the variable is fixed, optimized, constrained, the
    range for the optimization, etc."""

def updateVariables(self):
    """ Update the definitions for all variables, mainly the value
    """

def getVariable(self, fullName):
    """ returns a XmlVariable by its name, or None if not exists. The
    Fullname of a variable is modelName.parameterName.attributeName.
    """

def getConstrains(self):
    """ Return a list of XmlConstrain object."""

def setConstrain(self, target, expression):
    """ Add a new constrain. """

def getConstrain(self, fullName):
    """ return a XmlConstrain by its name, or None if not
    exists."""

def getModels(self):
    """ Return a list of XmlModel. """

def addModel(self, model):
    """ Add a new model. """

def getModel(self, modelName):
    """
        Return a XmlModel by its name, or None if there is
        no model of that name.
    """

def getModelNames(self):
    """ Return a list of the names of all the models. """

def toFile(self, fname):
    """ Store the xml string in the file. """

def fromFile(self, fname):
    """ restore the fitting from the file. """

```

The `parkUniFit.ParkUniFit` class is an abstract class, it simplifies the interface to build and access the model. Some of its important functions are:

```
class ParkUniFit(ParkFit):
```

```

def __init__(self, nparams = None):
    """
        constructor.
        nparams: a list of n integers, indicating
                the number of parameters for each model.
    """

def setDataSource(self, modelName, files):
    """ set a list of file names as the data source for the given
model"""

def getModel(self, name):
    """ Return a XmlModel by its name. """

def getModelNames(self):
    """ Return a list of string to specify the XmlModel. """

#####
# The following are some protected functions that the subclasses
# need to implement.
#####

def _getDataset(self):
    """ Return the dataset for the model. The subclass must
        implement this function.
    """

def _getData(self):
    """ Return the data for the model. The subclass must
        implement this function.
    """

def _getTheoryClassName(self):
    """ Return the name of theory class for the model. The
        subclass must implement this function.
    """

def _getModelTypeName(self):
    """ Return the name of the type that can be handled by ParkGUI
        client. subclass may implement this function so it can be
        used by GUI client applications.
    """

def _getModelNamePrefix(self):
    """ Return the prefix string to label the model for the
        fitting. The subclass may override this function. The
        model's name will be '%s%i' %(_getModelNamePrefix(), i),
        where i is the index of the model in the fitting.
    """

def _getParamNamePrefix(self):
    """ Return the prefix string to label the parameters in the
        model. The subclass may override this function. The
        parameter's name will be '%s%i' %(_getParamNamePrefix(),
        i), where i is the index of the model in the fitting.
    """

```

```
def _createParameters(self, model, count):  
    """ Create the parameters for the model. The subclass may  
        override this function. The default parameter will be used  
        if it return False  
    """  
  
def _getDefaultParameter(self):  
    """ Return the default parameter for the model if  
        self._createParameters() return False.  
        The subclass may override this function.  
    """
```

Currently, one subclass of `parkUniFit.ParkUniFit`, `parkGaussFit` class is implemented to handle the fitting of multiple Gaussian models. The details about its implementation will be discussed in the following chapter.

### 3 Gaussian Model

This chapter describes how to plug a new model (here is the Gaussian model) into the fitting service, and how to access the fitting service for this Gaussian model by script client application or GUI client application. Many models are similar to this Gaussian model. They can be implemented in the similar way as that of Gaussian model, and plugged into this PARK framework, including the fitting service framework, the script client application framework, and the GUI client application framework. All the source code for Gaussian model is implemented in the directory *park/theory/gauss*.

#### 3.1 Introduction

The Gauss model we are developing is to fit our data with multiple Gaussian functions:

$$y = \sum_{i=1}^n a_{0,i} * \exp[-((x - x_{0,i}) / \sigma_i)^2] \quad (1)$$

Each Gaussian function can be described by the parameters:  $a_0$ ,  $x_0$ ,  $\sigma$ . Once we have the number of Gaussian functions, and the parameters for the individual Gaussian function, the theoretical value can be calculated by (1), which can be fitted to our experimental data to get the optimized values of the parameters.

#### 3.2 Define Parameter

First we define the GaussParameter class, which has three members  $a_0$ ,  $x_0$ ,  $\sigma$ , It fully describes the Gaussian function:

```
#####
class GaussParameter(XmlParameter):
    """
        The parameters (x0,x0,sigma) for the Gaussian function:
        f(x) = a0 * exp(-[(x-x0)/sigma]^2),
        whose default values are :
        a0 = 1.0 , x0 = 0.0, sigma = 1.0
    """
#####

    def __init__(self):
        """ constructor. """
        super(GaussParameter, self).__init__()
        self.setDefault()
#####

    def _postParse(self): # set the default value
        """ set the default gaussian parameters if it doesn't
exists. """
        self.setDefault()
```

```
#####
def setDefault(self):
    setDefault(self, 'a0', GAUSSIAN_A0_DEFAULT)
    setDefault(self, 'x0', GAUSSIAN_X0_DEFAULT)
    setDefault(self, 'sigma', GAUSSIAN_SIGMA_DEFAULT)
#####
```

Notes: Normally you don't need to redefine your own parameter class which should inherit from `park.fit.XmlParameter`, but use the `XmlParameter` class directly. `XmlParameter` is a generic class which can hold many attributes. The attribute value can be accessed directly by its name. The advantage of self-defined `XmlParameter` is it can provide the default value for the attribute even the user doesn't assign a value for it. The following two code sections are nearly identical from the fitting service's point view:

```
pm = XmlParameter()
pm.a = 10.0
pm.x0 = 0.5
pm.sigma = 2.5
```

and

```
pm = GaussParameter()
pm.a = 10.0
pm.x0 = 0.5
pm.sigma = 2.5
```

The attribute name will be shown in the variable definition, which defines whether it is optimized, fixed, or calculated from a constrain expression. If the attribute will not be shown in the variable definition, use the '\_' started string as the attribute name.

Some times we have several different parameters to describe the model. For example, for reflectometry mode, we use the interface and layer to describe the model, for SANS model, we use different objects, such as the sphere, cylinder, or ellipsoid to describe the model. We can use '\_' attribute to label the type of parameters, such as:

```
spm = XmlParameter()
spm._type='Sphere'
spm.x0 = 0.1
spm.y0 = 0.5
spm.z0 = 1.5
spm.r = 8.0

cpm = XmlParameter()
cpm._type='Cylinder'
cpm.x0 = 0.1
cpm.y0 = 0.5
cpm.z0 = 1.5
cpm.h = 10.0
cpm.r = 5.0
```

In this case, the variables are ('x0', 'y0', 'z0', 'r') for sphere parameter `spm`, and

('x0', 'y0', 'z0', 'r', 'h') for the cylinder parameter cpm.

### 3.3 Define Theory

Once we have the parameters for the model, we can define class to perform the theoretical simulation. The theory class for the model should inherit from Theory class (defined in park/theory/parkTheory.py), which defines some basic interfaces used for the fitting service. They are:

```
XmlTheoryData & getFx():
    return a theory data object, mainly used for the visualization

double getObjectiveFx():
    return the objective function, used for optimization

double * getResidual():
    return the residual array, mainly used for the uncertainty
        calculation for least-square problem(Normally fitting is
        one of that least-square problem).

#####
class GaussTheory(Theory):
    """ A simple example to indicate the implementation of a theory.
        The data is one array x[i], and the output is y[i]:
        y(i) = sum[ao[k]*exp(-(x[i]-x0[k])**2/sigma[k]**2)] for k
    """
#####
    def __init__(self):
        Theory.__init__(self)

#####
    def _getFx(self, data, params):
        """ calculate the gaussian function. data is the data
            object return from the reduction data of dataset.
            The parameters are list of Gaussian parameter.
        """
        if data is None:
            return
        try:
            xdata = data[0].getData()
            array = XmlDataArray()
        except:
            xdata = data[0]
            array = None

        y0 = numpy.zeros(len(xdata), numpy.float64)
        for pm in params:
            dx =(xdata - pm.x0)/pm.sigma
            y0 += pm.a0 * numpy.exp(-dx*dx)

        if array is None:
            return (y0,)
        else:
            array.setData(y0)
            return (array, )
```

```

#####

def _getObjectiveFx(self, data, params):
    """ calculate the chisq.
    """
    dy = self._getResidual(data, params)
    return numpy.dot(dy, dy)
#####

def _getResidual(self, data, params):
    """ Return a vector to calculate norm2. It is required by
        leastsq() method to calculate the error bar of
        parameters.
    """
    try:
        xdata = data[0].getData()
    except:
        xdata = data[0]

    try:
        ydata = data[1].getData()
    except:
        ydata = data[1]

    y0 = numpy.zeros(len(xdata), numpy.float64)

    for pm in params:
        dx = (xdata-pm.x0)/pm.sigma
        y0 += pm.a0 * numpy.exp(-dx*dx)

    dy = y0 - ydata
    #print 'type', type(dy)
    return dy
#####

```

Normally, the Theory class may provide the following functions:

1. The function to calculate the chisq value: This function returns a float value and is implemented by function getObjectiveFx() or \_getObjectiveFx(). It is required for the optimization
2. The function to calculate the residual: This function returns a float array and is implemented by function getResidual() or \_getResidual(). It is required to calculate the uncertainty estimation using Levenberg-Marquardt algorithm.
3. The function to calculate the theoretical results: This function returns a list of DataObject objects and implemented by function getFx() or \_getFx(). It is required to visualize the simulation results.

The get\* functions are the interface to the outside, normally they call the corresponding \_get\* function to do the real calculation with the given data and parameters.

For most subclasses of ParkTheory, they may only need to implement the `_get*` functions, and normally the `_getObjectiveFx()` is the norm of `_getResidual()`.

### 3.4 Define Dataset

Dataset class provides the experimental data for the fitting, which comes from underneath raw data represented by one or multiple Data object. Sometimes the Dataset may have no Data objects, for example in the simulation. Dataset may also provide some other information, such as the meta data for the model, some information for the simulation when there is no experimental data to fitting. Basically, the theory class should be interactive with Dataset object only, use the reduction data and/or meta data that the Dataset object provides to do the theoretical simulation.

Partial implementation of the Dataset class for Gaussian model is shown in the following:

```
# class name of dataset for Gaussian model
GAUSS_DATASET_CLASSNAME = 'gaussDataset.GaussDataset'

#####
class GaussDataset(XmlDataset):
    """ The dataset for Gaussian model """
    #####
    def __init__(self):
        super(GaussDataset, self).__init__()
        self.setClassName(GAUSS_DATASET_CLASSNAME)
    #####

    def _getNodeObject(self, nodename):
        if nodename == DATA_TAG:
            return GaussData()
        else:
            return super(GaussDataset,
                          self)._getNodeObject(nodename)
    #####

    def _checkXmlReductionData(self, reductionData):
        """ Check to make sure there is a valid reduction data """
        # The reduction data are [X, Y], both are array.
        cnt = len(reductionData.getXmlDataArray())
        if cnt < 2:
            for ind in xrange(2-cnt):
                reductionData.add(XmlDataArray())
    #####

    def _updateXmlReductionData(self):#, datalist):
        """ Update the reduction data for the whole dataset. """

        try:
            dataarray = []
            for data in self.getXmlData():
                dataarray.append(data.getDataList())
            if len(dataarray) >= 1:
```

```

        mydata = joinData(dataarray)

        if mydata is None:
            return [None, None]
        else:
            return [mydata[0], mydata[1]]
    else:
        return [None, None]
except:
    print 'exception:', traceback.format_exc()
    return [None, None]

```

#####

The subclasses of XmlDataset may need to implement the following two functions:

**void \_checkXmlReductionData(reductionData):**

This function checks reductionData, an XmlReductionData object of the dataset, and makes sure that it has enough room to contains the data return from the following function `_updateXmlReductionData()`

**[ double \* ] \_updateXmlReductionData():**

This function will update the reduction data of XmlDataset when its contents have been changed, such as the associated meta data, or the underneath data are changed. For this dataset class, the data may from multiple files, so these multiple data should be joined together. It return a list of float array, or matrix, etc., and will be used as the contents of the XmlReductionData in the model. This function can also considered as the simple reduction procedure to process the raw data.

### 3.5 Define Data

Data objects are contents of Dataset class to handle the raw data. The Data class provides all the pre-processing procedures to deal with the data source, and get the available reduction data for the Dataset for further processing. For example, if the data source is the local file, the Data class needs implement the function to read the data from a file with the format it knows. If the data source is the remote data, the Data class may need to load the data remotely. The Data class also need to handles the meta data associated with the raw data. These functions are specified for the specific data format and data processing. However, the associated Dataset object only interactive with the reduction data that the Data object provides.

Partial implementation of the Dataset class for Gaussian model is shown in the following:

```

#####
GAUSS_INSTRUMENT_DATA_TAG = 'gauss'
GAUSS_DATA_CLASSNAME = 'gaussData.GaussData'

```

```

#####
class GaussData(XmlData):
    """ Gaussian data: two columns: X - Y """

#####

    def __init__(self):
        """ Constructor. """
        super(GaussData, self).__init__()
        self.setClassName(GAUSS_DATA_CLASSNAME)
        ## the raw data, we only have (x, y) pair of data
        self._xx0 = None
        self._xy0 = None

#####

    def __postParse(self):
        """ Provide the default value for the raw data. """
        if not hasattr(self, '_xx0'):
            self._xx0 = None
        if not hasattr(self, '_xy0'):
            self._xy0 = None

#####

    def __getNodeObject(self, nodename):
        if nodename == GAUSS_INSTRUMENT_DATA_TAG:
            return GaussInstrumentData()
        else:
            return super(GaussData,
                          self).__getNodeObject(nodename)

#####

    def __checkXmlReductionData(self, reductionData):
        """ Check the validity of reduction portion. """
        #reduction = super(GaussData, self).checkReductionData()

        cnt = len(reductionData.getXmlDataArray())
        if cnt < 2:
            for ind in xrange(2-cnt):
                reductionData.add(XmlDataArray())

#####

    def update4Source(self):
        """ Update the reduction data due to change of data source. """
        #print 'source:', self.source
        try:
            [self._xx0, self._xy0] = self._readRawData()
        except:
            #print 'except:', traceback.format_exc()
            self._xx0 = None; self._xy0 = None
        #print 'self.data', self._xx0, self._xy0

        self.update4Meta()

#####

    def update4Meta(self):
        """

```

```

        Update the reduction data due to change of meta data.
        """
        data = self.checkXmlReductionData().getXmlDataArray()
        data[0].setData(self._xx0)
        if self._xy0 is None:
            data[1].setData(None)
        else:

            data[1].setData(self._xy0*self.getXmlInstrumentData().scale
                )
#####

def _readRawData(self):
    """ Return the original Gauss data array: [X,Y]
        """

    filename = getattr(self,DATA_SOURCE_TAG)

    return readAsciiData(filename)
#####

```

The subclasses of XmlData may need to implement the following two functions:

**void \_checkXmlReductionData(reductionData):**

This function similar to the **\_checkXmlReductionData(reductionData)** of XmlDataset, it checks the reductionData, an XmlReductionData object of the data, and makes sure that it has enough room to contains the data return from the following functions **update4Source()** and **update4Meta()**

**void update4Source():**

**void update4Meta():**

These functions will update the reduction data of XmlData when its source has been changed (by **update4Source()**) or its meta data has been changed (by **update4Meta()**). The reduction processing may be applied in these functions.

The data for Gaussian model fitting is from the local file, the GaussData also provide function **\_readRawData()** to read the raw data from local file system, and return a list of data. These data may be raw data, or the reduction data after the primary pre-processing. They may be cached to increase the performance by avoid the repeatedly reading the raw data, and pre-processing the raw data if the reduction data are cached.

### 3.6 Define MetaData

For some data fitting, the experimental data have some meta data associated with it. These meta data may indicate the experimental conditions, or some other conditions, and affect the reduction data for the fitting or simulation. They be handled in the reduction stage.

For the Gaussian model, one data set may be composed of multiple Gauss data, each is the different range of x value. These multiple data should be joined together before the fitting procedure. For these Gaussian model, the meta data is very simple, just the scaling factor for the y value. The meta data class for Gaussian model is GaussInstrumentData, which has only one property: scale.

```
class GaussInstrumentData(XmlAttribute):
    """ Gaussian Instrument data. """
    #####
    def __init__(self):
        super(GaussInstrumentData, self).__init__(
            GAUSS_INSTRUMENT_DATA_TAG)
        self.setDefault()

    #####
    def _getSelfObject(self):
        """ return a copy of current object. """
        return GaussInstrumentData()

    #####
    def _postParse(self):
        self.setDefault()

    #####
    def setDefault(self):
        SetDefault(self, 'scale', GAUSS_A0 )
    #####
```

For some complicated models, their meta data may have more properties. Its effect on the reduction data of XmlData is shown in the function `update4Meta()` of class XmlData.

Until now, we have defined the data structure for the Gaussian model, which includes

- the GaussParameter class for the parameter,
- the GaussTheory class for the theoretical simulation,
- the GaussDataset class for the dataset to provide reduction for the theory, and
- the GaussData class to provide function to handle the raw data, and provide the reduction data for the GaussDataset object.
- the GaussInstrumentData class to provide the meta data information about the Gaussian data.

All of these classes may also be needed to build a new model.

Place these implements to where the fitting service can access, current it is under the directory of theory/gauss/. With these implementations and the fitting service, we can carry out the data fitting for Gaussian model, either using the script client applications, or using the GUI client applications. We will start from the script applications first.

### 3.7 Script client application

Now that we have the fitting service for Gaussian model, we can submit the request for the fitting, which needs to be done by two steps:

- Build the fitting for the fitting service

- Send the request to the PARK server.

In chapter 2, the details about the communication have been discussed. Here we will learn how to build the fitting for the fitting service.

The fitting for the fitting service can be built directly by using the XmlFitting and its associated classes. However, these classes are designed to fit the generic requirement for data fitting, and are a little more complicated to build the fitting with simple model, such as the Gaussian model. In order to simplify the user interface, the adapter classes can be developed to provide simple user interface so the user can build the Gaussian model for data fitting easily. The above mentioned ParkFit and ParkUniFit classes are those kinds of adapter classes to build the fitting by script application. They are still generic, and can be further refined to fit the specific model. These specified adapter classes are more easily to use, but they are also closely associated with the given model.

For the fitting of Gaussian model, we define a adapter class, ParkGaussFit, whose implementation is shown as:

```

GAUSS_PARAM_NAME_PREFIX = "g"

GAUSS_THEORY_CLASS_NAME = "gaussTheory.GaussTheory"

GAUSS_GUI_TYPE_NAME = 'gauss'
#####
""" The adapter class to simplify the interface between the script
    application and the framework of fitting service.
"""
#####

class ParkGaussFit(ParkUniFit):
    """
        The subclass class of ParkUnitFit where all the models are
        Gaussian model.
    """
    #####
    def __init__(self, nparams=None):
        """
            constructor.
            n: number of Gaussian models
            nparams: a list of n integers, indicating
                    the number of Gaussuan parameters.
        """
        super(ParkGaussFit, self).__init__(nparams)
        #####

    def _getDataset(self):
        """ Return the dataset for the model. The subclass must
            implement this function.
        """
        return GaussDataset()
    #####

```

```

def _getData(self):
    """ Return the data for the model. The subclass must
        implement this function.
    """
    return GaussData()
#####

def _getTheoryClassName(self):
    """ Return the name of theory class.for the model. The
        subclass must implement this function.
    """
    return GAUSS_THEORY_CLASS_NAME
#####

def _getDefaultParameter(self):
    """ Return the default parameter for the model.
        The subclass may override this function.
    """
    return GaussParameter()
#####

def _getParamNamePrefix(self):
    """ Return the prefix string to label the parameters in the
model.
        The subclass may override this function.
    """
    return GAUSS_PARAM_NAME_PREFIX
#####

def _getModelTypeName(self):
    """ Return the name of the type that can be handled by ParkGUI
client.
        subclass must implement this function.
    """
    return GAUSS_GUI_TYPE_NAME

```

---



---

Once this adapter class is defined, we can more easily to define a Gaussian model fitting. For example,

```
fit = ParkGaussFit()
```

defines a fitting with one Gaussian model , while that Gaussian model has only one Gaussian parameter. The attributes of that Gaussian parameter have the individual default values specified by GaussParameter class. Besides that, that fitting has the default optimizer, the default variable definitions (all the attributes will be optimized), and default constrains (No constrains). The model's name is called 'Mod0', and the Gaussian parameter's name is called 'g0'. We can access the attribute's value by:

```

modelNames = fit.getModelNames()
model = fit.getModel(modelNames[0])
model.g0.x0 = 10.0
model.g0.sigma = 0.5

```

etc. What we need further is to assign the data source for the model, which also can be done by calling the `setDataSource(modelName, dataFiles)` function:

```
dataFiles = ['gauss0.dat', 'gauss1.dat']
fit.setDataSource(modelNames[0], dataFiles)
```

Then we can use this fit object as the input to request the Gaussian model fitting directly. This will be shown soon.

Furthermore, we can build a fitting with multiple Gaussian models and each Gaussian model has multiple Gaussian parameters. For example,

```
p0 = [2, 3]
fit = ParkGaussFit (p0)
```

will build a Gaussian fitting with two Gaussian models, the first Gaussian model is called 'Mod0' and has two Gaussian parameters, whose name are assigned as 'g0' and 'g1', while the second Gaussian model is called 'Mod1' and it has three Gaussian parameters, whose names are assigned as 'g0', 'g1', 'g2'. Then you can use the same procedure to manipulate the data source, the attribute value. If you like, you can use `getOptimizer()` function to change the default setting for the optimizer, `getVariable(varName)` to change the optimization properties of that variable, and `setConstrains(varname, cons_expr)` to set the constrains.

Once we know how to build the fitting and send the request to the server, we can easily to build a scrip application to run the Gaussian model fitting. The following is a example for that, which is implemented in `script/parkGaussClient.py`.

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

import os, time, sys
import traceback
#####

from parkFitClient import ParkFitClient
from parkGaussFit import ParkGaussFit

from scriptUtil import SetGaussEnviron

SetGaussEnviron()

#####

GAUSS_FILES = ['gauss0.dat',
               #'gauss1.dat'
               ]
DATA_DIR = '../examples/gauss'
```

```

def buildGaussianFitting():
    ## two gaussian parameters in one Gaussian model
    p0 = [2]
    dataFiles = [os.path.join(DATA_DIR, fname)
                 for fname in GAUSS_FILES ]

    fit = ParkGaussFit(p0)
    modelNames = fit.getModelNames()
    fit.setDataSource(modelNames[0], dataFiles)
    model = fit.getModel(modelNames[0])

    # set the initial values.
    # for first gaussian parameter
    model.g0.a0 = 15.0
    model.g0.x0 = 0.4
    model.g0.sigma = 0.3

    # for second gaussian parameter
    model.g1.a0 = 29.0
    model.g1.x0 = 1.5
    model.g1.sigma = 0.4

    return fit

#####

"""
    An example to show how to write a script-based service client
    application: build models for the fitting, submit the request,
    wait for the reply.
"""
#####

def main1():
    """ An example for Gaussian fitting """
    client = ParkFitClient()

    req = buildGaussianFitting()
    client.submit(req, 1)

    while (not client.isCompleted()):
        job = client.getJob()
        # job is type of ParkFit
        print 'job', job
        print 'job overview:', job.getFittingResultOverview()

    del client

def main5():
    """ An example for Gaussian fitting """
    client = ParkFitClient()

    req = buildGaussianFitting()
    client.submit(req, 5)

    while (not client.isCompleted()):

```

```

    job = client.getJob()
    # job is type of ParkFit
    print 'job', job
    print 'job overview:', job.getFittingResultOverview()

del client
#####

if __name__ == '__main__':

    main1()
    main5()

#####

```

The main1() program will request to run the given Gaussian fitting just one time, while the main5() program will request to run it for five times. You can run it in background or somewhere and leave it alone. Once it finishes, you will get the fitting results. The various model building methods and communication mode can be combination to fit the client user's requirement.

Once again, the development of the adapter classes is not necessary for the script client applications. However, they can be designed to be closely associated with the specific models and significantly simplify the interface for the end script client applications. These adapter classes are highly dependent on the underneath model. PARK provides some generic adapter classes to be inherited. It is the model developer's responsibility to design and develop these kinds of adapter classes.

It is a little different to build fitting under GUI environments, where the complexity to build fitting by directly using XmlFitting is hidden by the provided GUI components. The user uses the GUI components to build the fitting and is invisible to the underneath, real objects' creation. The adapter classes are totally unnecessary under GUI client applications. The GUI client applications need a GUI framework so the used defined GUI components can be plugged into it, and build the fitting. This is what the PARK GUI client provides.

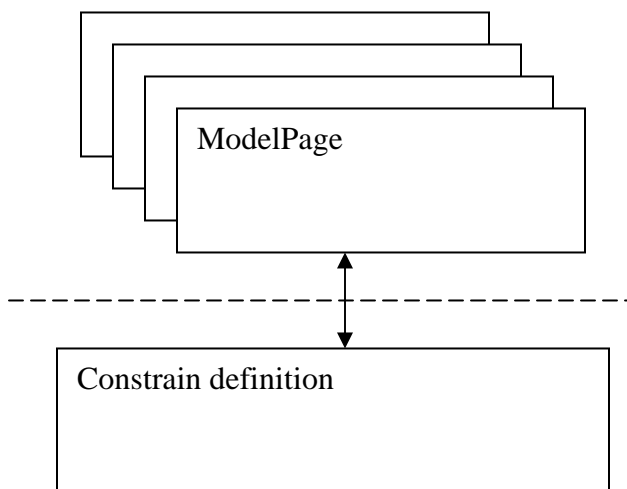
## 4. GUI framework for fitting client

### 4.1 Fitting GUI client

The service is running at the server. The client needs to send the service request to the server to ask for the service not matter what kind of method it uses. The client can use a script file, or a GUI application to send the request.

PARK release includes a Fitting GUI client which corresponds to the provided fitting service. Like the fitting service is pluggable, the fitting GUI client is also a pluggable framework. The model developer can implement the GUI representation of their models, and plug them into this fitting GUI client. The framework provides the functions, such as sending the fitting service request, receiving the fitting results, wiring the communications among different GUI components.

Basically, the model GUI developer needs to provide a model page GUI component. It is plugged into fitting GUI client framework, provides an interface for the user to modify and view the model interactively. The fitting GUI client framework also provides a common constrain GUI components to define the variable definition and constrains for the variables. The constrain GUI component will communicate with the model page. The low level of interaction in Fitting GUI framework is shown in Fig. 2.



From the model GUI developers' point view, they can implement their own FittingModelPages, which are just some kind of GUI panel components, and plug them into the fitting GUI client framework. This option gives the developers the greatest freedom for the implementation. However, the developers need to handle the communication among its own components, and to the outside constrain components. The user defined FittingModelPage needs to be a wx.Panel and at least implement the follow interfaces:

```
void SetModel(XmlModel & model)
    ""Set the model for the page. ""
```

```

XmlModel & GetModel()
    """Return the model for the page"""

void UpdateViewer()
    """ Update the page for the model structure is changed"""

void UpdateParameterViewer(string paramName, float value):
    """ Update the page for one parameter's value is changed"""

void Enable(bool enable):
    """ Enable or disable the page"""

void SetEditMode(bool editMode)
    """ Set the page editable or not. """

```

The alternative option is to use the pre-defined fitting client GUI implementation, which implement the framework for the fitting service request, including the corresponding GUI components for fitting service data structure, the communication among the GUI components, the interface for individual GUI components. The UML diagram for these predefined fitting client GUI components is shown in Fig. 3.

Considering the data structure of fitting service, this abstract model page, defined by the abstract class `FittingModelPage`, is separated into three GUI components to specify the individual underline data structure:

**Dataset panel:** Defined by the abstract class `FittingDatasetPanel`, view and edit the dataset associated with the model, and may be the associated theoretical results

**ResidualPanel:** Defined by the abstract class `FittingResidual`, view the residual of the specified model. This is the optional part, it is unnecessary for some cases, such in the simulation.

**Model builder:** Defined by the abstract class `FittingModelBuilder`, the GUI representation of the model. Mainly, the user uses this component to change the value and number of the associated parameters interactively.

As a subclass of `FittingModelPage`, it must implement the following functions:

```

FittingDatasetPanel & _getDatasetPanel():
    """ Return a panel to view/edit the dataset for the model. """

FittingModelBuilder & _getBuilderPanel():
    """ Return a panel to build and view the model. """

FittingResidual & _getResidualPanel():
    """ Return a panel to view the residual of the model. """

```

More details, the dataset panel may include the following two GUI components:

**Dataset view panel:** Defined by the abstract class `FittingDatasetViewerPanel`, view the reduction data of the data set and the theoretical data of the model.

Dataset editor panel: Defined by the abstract class `FittingDatasetEditorPanel`, edit the contents of the dataset, including add/remove the data, view the original data, view/edit the associated meta data. It may have the GUI component to view/edit the data, and the GUI component to view/edit the meta data.

So as a subclass of `FittingDatasetPanel`, it must implement the following interfaces:

```
FittingDatasetViewer & _getDatasetViewer():  
    """ Return a panel to view the reduction data of the dataset  
    and theory data of the model."""  
  
FittingDatasetEditor & _getDatasetEditor():  
    """ Return a panel to edit the dataset of the model """  
  
XmlDataset & _getDataset(self):  
    """ Return a object of the subclass of XmlDataset to represent  
    the dataset of the model """
```

Here, the `FittingDatasetViewer` is any plottable GUI component that implements the following interface to update its contents:

```
void SetModel(XmlModel & model)
```

The `FittingDatasetEditor` is a predefined GUI component to edit the contents of the dataset of the model, which includes two components:

`FittingDatasetDataPanel`: Edit the contents of the data included in the dataset, such as add and remove data from the dataset, save and load `XmlDataset` from a file, etc. A `ParkDatasetEvent` event should be emitted when the data contents of the dataset is changed.

`FittingDatasetMetaPanel`: Edit the contents of the meta data associated with the data of the dataset. A `ParkDatasetEvent` event should also be emitted when the meta data contents of the dataset is changed.

The subclass of `FittingDatasetEditor` needs to implement the following interfaces:

```
FittingDatasetDataPanel & _getDatasetVEditor():  
    """Return a Panel to edit data contents of the dataset. """  
  
FittingDatasetMetaPanel & _getMetaPanel():  
    """Return a panel to edit the meta data associated to a data of  
    the dataset. """
```

`FittingDatasetDataPanel` implements the functions to edit the included data contents of the dataset. It should be subclassed to implement the following interfaces:

```
XmlData & _getData()  
    'Return an object of subclass of XmlData representing the data.'  
  
XmlDataset & _getDataset():
```

```
'Return an object of subclass of XmlDataset representing the dataset for the model.'
```

```
void _updateViewer(self):  
'Update the viewer of the data.'
```

The subclass of [FittingDatasetDataPanel](#) may also call the following function to send the notifying event to the outside world that its data contents have been changed:

```
void _fireEvent(style=DATASET_UPDATE):  
""" Fire the ParkDatasetEvent event."""
```

Similarly, [FittingDatasetMetaPanel](#) implements the functions to edit the meta data associated with one data of the dataset. Its subclass should implement the following interface:

```
void _updateViewer(self):  
'Update the viewer of the data.'
```

and call the following function to send the notifying event to the outside world that its meta data contents have been changed:

```
void _fireEvent():  
""" Fire the ParkDatasetEvent event."""
```

The synchronization between the data and associated meta data is handled by the corresponding [FittingDatasetEditor](#).

The [FittingModelBuilder](#) define the GUI representation of the model. It provides the interface for the implementation of the real model. The implementation of real model builder can be very simple, or very complicated depending on the complexity of corresponding model. The subclasses need to implement the following interfaces:

```
String & _getTheoryName(self):  
""" Return the class name of the theory associated with the model"""  
  
void UpdateViewer(self):  
""" Update when the model or its content is changed."""  
void UpdateParameterViewer(self, names, value):  
""" Update when the specified attribute of the model is changed. """
```

In the implementation, the following functions can be called to send the notifying event that either the contents of the model are changed ([ParkModelEvent](#) is emitted), or the value of the specified attribute of the model is changed ([ParkParameterEvent](#) is emitted):

```
void _fireParameterEvent(self, names, value):  
""" fire an event to notify that the value of a specific attribute of the model is changed """
```

```

void _fireModelEvent(self, style):
    """ fire an event to notify that the contents of the
        model is changed """

```

ParkParameterEvent is a special case of ParkModelEvent, where only the value of one attribute of the model is changed. The developer may only to update a small portion of the panel instead to update the whole panel, which may speed the update of the panel for the complexity model.

The developers can also use the following functions to initialize the contents of the model, which are called in that sequential order in `FittingModelBuilder`:

```

void _init_ctrls()
    """ Create the controls for the model builder. """
void _init_sizers()
    """ Create the sizers and put the controls in them. """
void _initContextMenu()
    """ Return the context menu for the model builder. """
void _bind_ctrls()
    """ Connect the event handlers to the GUI components. """

```

The developers can also use the following functions to initialize the contents of the model, which are called in that sequential order in `FittingModelBuilder`:

## 4.2 GUI client applications

As mentioned in Chapter 2, PARK GUI client provides a pluggable framework to plug the user defined GUI components. That GUI framework defines the layout of the GUI components, provides the common GUI components to define the variable definition and constrains, provides the communication between different GUI components, sends the request to the server, updates the results from the server, save the results, etc. It is a good alternative to use this GUI client framework to communication with PARK server, especially when the fitting is not too complicated.

Under PARK GUI client framework, user only needs to provide GUI components to edit the individual model. From the highest-level, the user only needs to implement a ModelPage GUI component, register it to a specific mode, and plug into PARK GUI client framework. Furthermore, considering the data structure for fitting service, PARK GUI client framework provides the framework for individual GUI component for each component of the model, which includes the underneath event communication, and the individual abstract GUI component classes, which are:

- FittingModelPage panel: A GUI component to show and edit the individual model. It contains the ModelBuilder and DatasetEditor panels.
- FittingModelBuilder panel: A GUI component to show the GUI representation of the model. It also provides interactive function for the user to edit the parameter contents of the model, such as add a parameter, remove a parameter, change the parameter's values, etc.

- FittingDatasetEditor panel: A GUI component to show the reduction data for the fitting, and provides a interactive function for the user to edit the underneath data, such as add/remove a data for the dataset, change the meta data for the model or the data. It contains a DatasetViewer panel.
- DatasetViewer panel: A GUI component to show the reduction data from the experiment for the fitting. It may also have the theoretical data from the simulation.
- DatasetData panel: A GUI component to show the reduction data for the individual data of the data set. It also provides the interactive functions so the user can iterate all the data of the dataset, add and remove the given data.
- DatasetMeta panel: A GUI component to show the meta data for the associated data. The user can edit the meta's parameter.

### 4.2.1 Model Page

The implementation of ModelPage GUI component for Gaussian model is very simple, it is shown in the following:

```
#####
class GaussModelPage(FittingModelPage):
#####

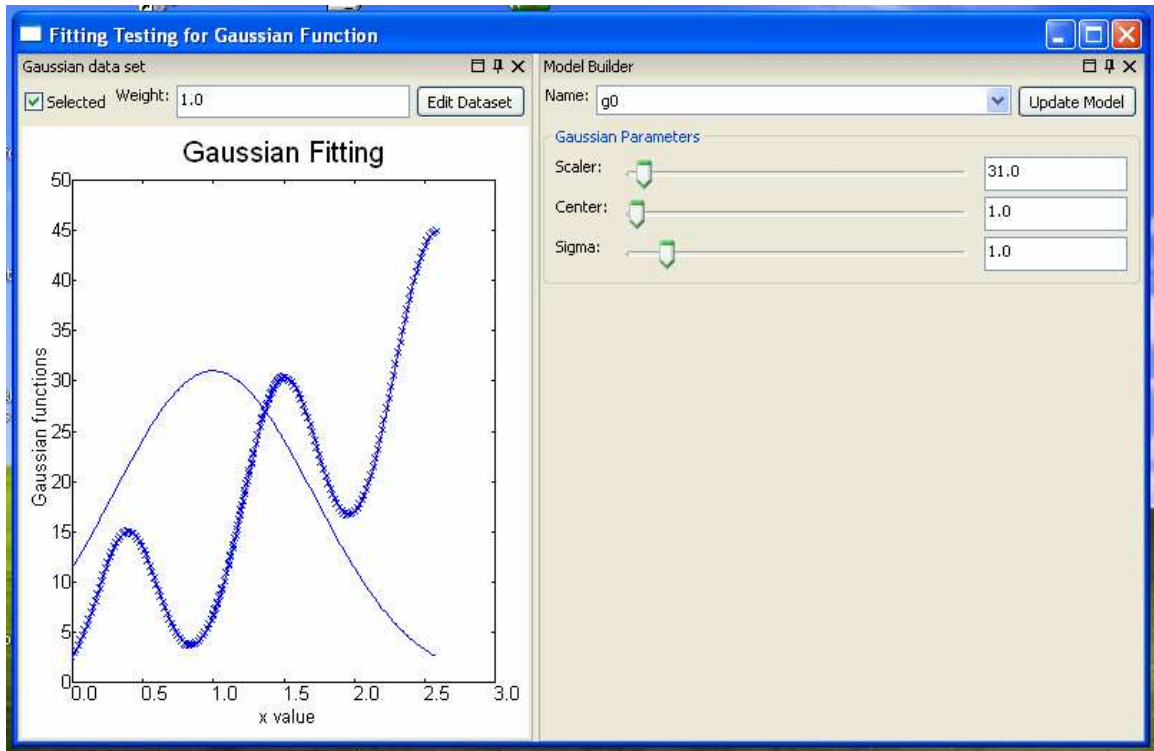
    def _getDatasetPanel(self):
        ## dataset viewer
        return GaussDatasetPanel(self)
#####

    def _getBuilderPanel(self):
        ## model builder for the fitting
        return GaussModelBuilder(self)
#####

    def _getDatasetTitle(self):
        return GAUSS_DATASET_TITLE
#####

    def _getModelTitle(self):
        return GAUSS_MODEL_TITLE
#####
```

In this class, we must implement the function `_getDatasetPanel()` which return a GUI component to view and edit the dataset of the model, and the function `_getDatasetPanel()` which will return a GUI component to view and edit the model. We may also override the functions `_getDatasetTitle()` and `_getModelTitle()`, which return strings to label the model builder panel and dataset panel, individual. The following is the figure for this model page. On the left is the dataset panel to edit and view the dataset, while on right is the model builder panel to view the model, edit, view and iterate its parameters.



#### 4.2.2 Model Builder

The ModelBuilder implementation is a little more complicated, because it total depends on the underneath model. The details about its implementation is in `parkAui/builder/gauss/gaussModelBuilder.py`, and the following is the figure to run this model builder.



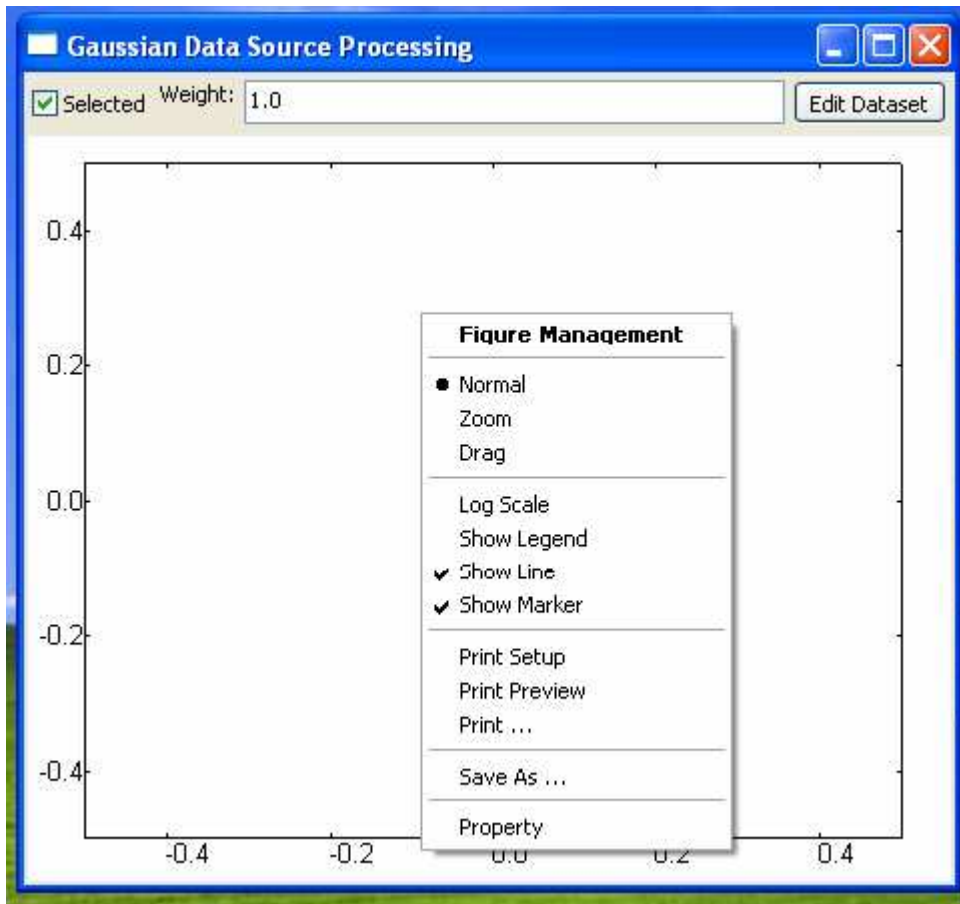
This model builder provides the following basic operations on the model:

- Add or remove a Gaussian parameter by its context menu
- Change the attribute value by the slider or input field, which is Scaler, Center, or Sigma attribute of the Gaussian parameter.
- Iterate the parameters of the model by the upper “Name” pull-down list, which is not shown here.

Depends on what kind of model will be built, this ModelBuilder panel can be very complicated, or as simple as this one for Gaussian model. To simplify the developing of the model builder, an abstract class, FittingModelBuilder class is defined. This abstract class defines some basic interfaces for the model builder to communicate with the outside. Some of its important functions are described in chapter 2.

### 4.2.3 Dataset Representation

On the left side of model page is the dataset panel, which mainly used to show the reduction data for the data, the theoretical simulation may also be shown if its calculation is not so expensive. There are two steps to build the dataset for a model. The above seen figure is the front end representation of the dataset. The underneath data will shown by clicking the “Edit Dataset” button, which will popup a dialog to edit the underneath data and the associated metadata for the dataset.



The implementation of this dataset panel for Gaussian model is also very simple, which is shown in the following:

```

class GaussDatasetPanel(FittingDatasetPanel):
    """ The panel to show fitting data and theoretical results. """
    #####

    def __init__(self, parent, id=-1, pos=wx.DefaultPosition,
                 size=DEFAULT_PANEL_SIZE, style=wx.TAB_TRAVERSAL,
                 name='fitting data source'):
        super(GaussDatasetPanel, self).__init__(parent=parent,
                                                id=id, pos=pos, size=size, style=style, name=name)
    #####

    def _getDatasetViewer(self):
        viewer = GaussDatasetViewerPanel(self)
        viewer.SetXLabel(X_LABEL)
        viewer.SetYLabel(Y_LABEL)
        viewer.SetTitle(TITLE)
        return viewer
    #####

    def _getDatasetEditor(self):
        return GaussDatasetEditor(self)
    #####

    def _getDataset(self):
        return GaussDataset()
    #####

#####
class GaussDatasetViewerPanel(PlotPanelF):
    #####

    def __init__(self, parent, id=-1, pos=wx.DefaultPosition,
                 size=DEFAULT_PANEL_SIZE):
        super(GaussDatasetViewerPanel, self).__init__(
            parent, id, pos, size)
    #####

    def SetModel(self, model):
        if model is None:
            self.Clear()
            return

        dataset = model.getXmlDataset()
        if dataset is None or \
            dataset.getXmlReductionData() is None:
            self.Clear()
            return

        ## return [X, Y] for Gaussian data
        gData = dataset.getXmlReductionData().getXmlDataArray()

        if (len(gData) < 2):
            self.Clear()

```

```

        return

    if (gData[0].getData() is None) or \
        (len(gData[0].getData())<=0 ):
        self.Clear()
        return

    viewdata = [gData[0].getData(), gData[1].getData(), None]

    try:
        viewdata.append( model.getXmlTheoryData().
            getXmldataArray()[0].getData())
    except:
        viewdata.append(None)

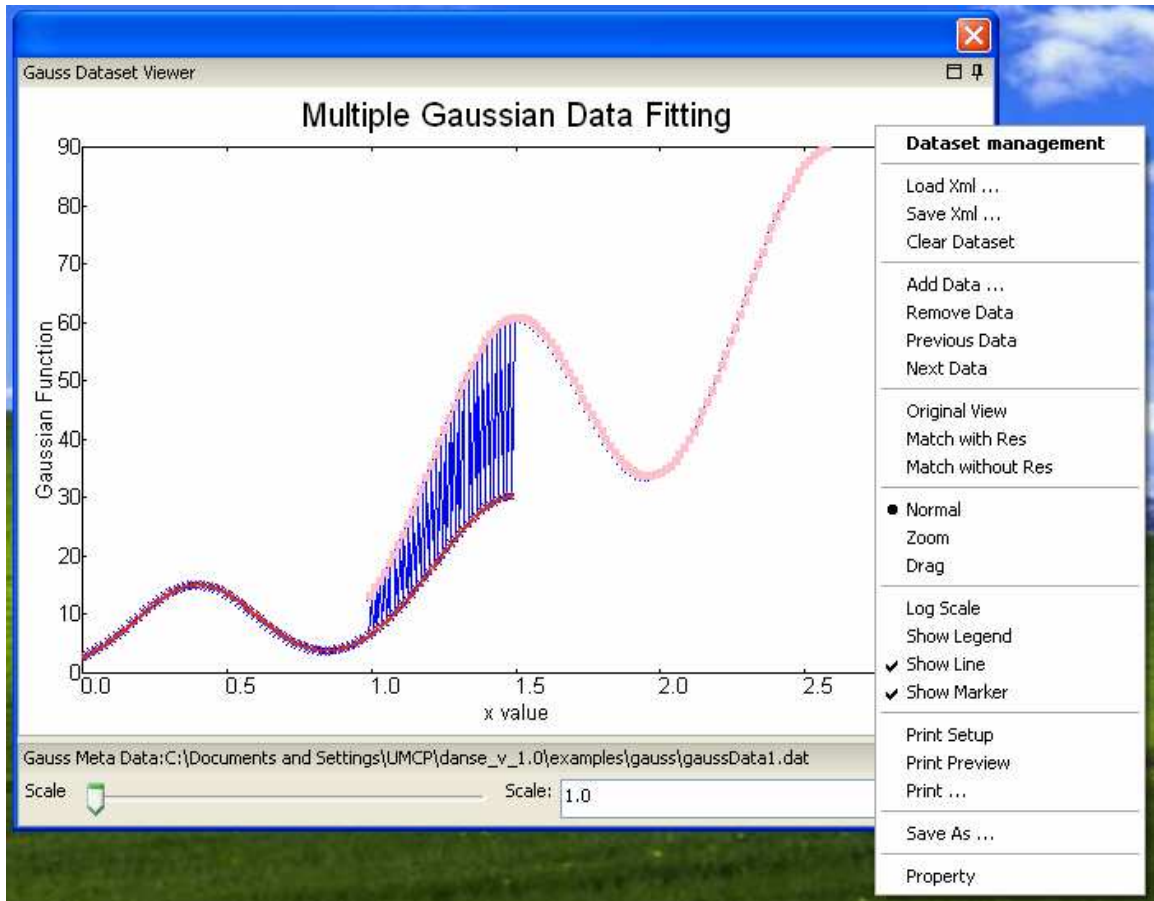
    self.SetData(viewdata)
#####

```

For the dataset panel, the main part is the GUI component to show the reduction data and/or the simulation data. This is the one that returns from the function `_getDatasetViewer()`, Another one is the dialog to edit the underneath data for the dataset, which returns from the function `_getDatasetEditor()`. Besides these two functions, it also need to return a `XmlDataset` object to represent the dataset that the panel is handling. The subclasses of `FittingDatasetPanel` must implement all these three functions. The GUI component to view the reduction data and simulation of the model is implemented by `GaussDatasetViewerPanel` class, which is just a GUI plotting component to show the data. It should update its window in its `SetModel()` function. PARK also provides some commonly used plotting GUI components under `parkAui/plot/` directory, most of them are for 1D data plotting.

#### 4.2.4 Data Editor

The next import GUI component is the one to edit the data and associated meta for the dataset, which is implemented by `GaussDatasetEditor` classes for Gaussian model and is shown in the following figure:



The DatasetEditor panel is also composed of two parts: The top one shows the reduction data for the individual data, while the bottom one shows its associated meta data. The top one also needs to provide some interactive functions so the user can edit and iterate through the underneath data, such as add and remove the data, go to the next or previous data, etc. The bottom one needs to provide function to change the value of the meta data, and notify the top one to update its windows when the meta data is changed.

The implementation of GaussDatasetEditor is also very simple, and is shown as the following:

```
#####
class GaussDatasetEditor(FittingDatasetEditor):
    def __init__(self, parent, id=-1, pos = wx.DefaultPosition,
                 size=DEFAULT_DIALOG_SIZE):
        super(GaussDatasetEditor, self).__init__(parent,
                                                  id=id, pos=pos, size=size)

#####

    def _getDatasetVEditor(self):
        return GaussDatasetDataPanel(self)

#####
```

```

def _getDatasetVEditorTitle(self):
    return VIEWER_TITLE
#####

def _getMetaPanel(self):
    return GaussDatasetMetaPanel(self)
#####

def _getMetaTitle(self):
    return META_TITLE
#####

```

This class inherits from `FittingDatasetEditor`, which already defines the underneath communication between the top data editing panel and the bottom meta data editing panel. The subclasses of `FittingDatasetEditor` must implement the `_getDatasetVEditor()` and `_getMetaPanel()` functions, which return a GUI component to view and edit the data and meta data individually. The subclasses may also override the functions `_getDatasetVEditorTitle()` and `_getMetaTitle()` to return the strings to label these two components.

The implementation of the data editing panel and meta data editing panel depends on the underneath data and meta data that they are handling. Their common interface have been described in chapter 2. The more details about their implementation can be found in `parkAui/builder/gauss/ gaussDatasetDataPanelP.py` and `gaussDatasetMeta.py`.

### 4.3 Plug GUI components

Once we have the model page GUI component for the model, we need to register it with the PARK client GUI framework. Currently, this registration is done manually. Edit the file `parkAui/common/fittingConfig.py`, import the model page component into that file, add the label name for that component in `AVAILABLE_MODELS`, and `MODELS`, then the GUI components for this model can be available from PARK GUI client applications. The configure file will be used to replace this registering procedure later. For example, the following lines in `fittingConfig.py` will register the Gaussian model and its GUI components into PARK GUI client frame:

```

##### ModelPager #####
from builder.gauss.gaussModelPage import GaussModelPage
#####

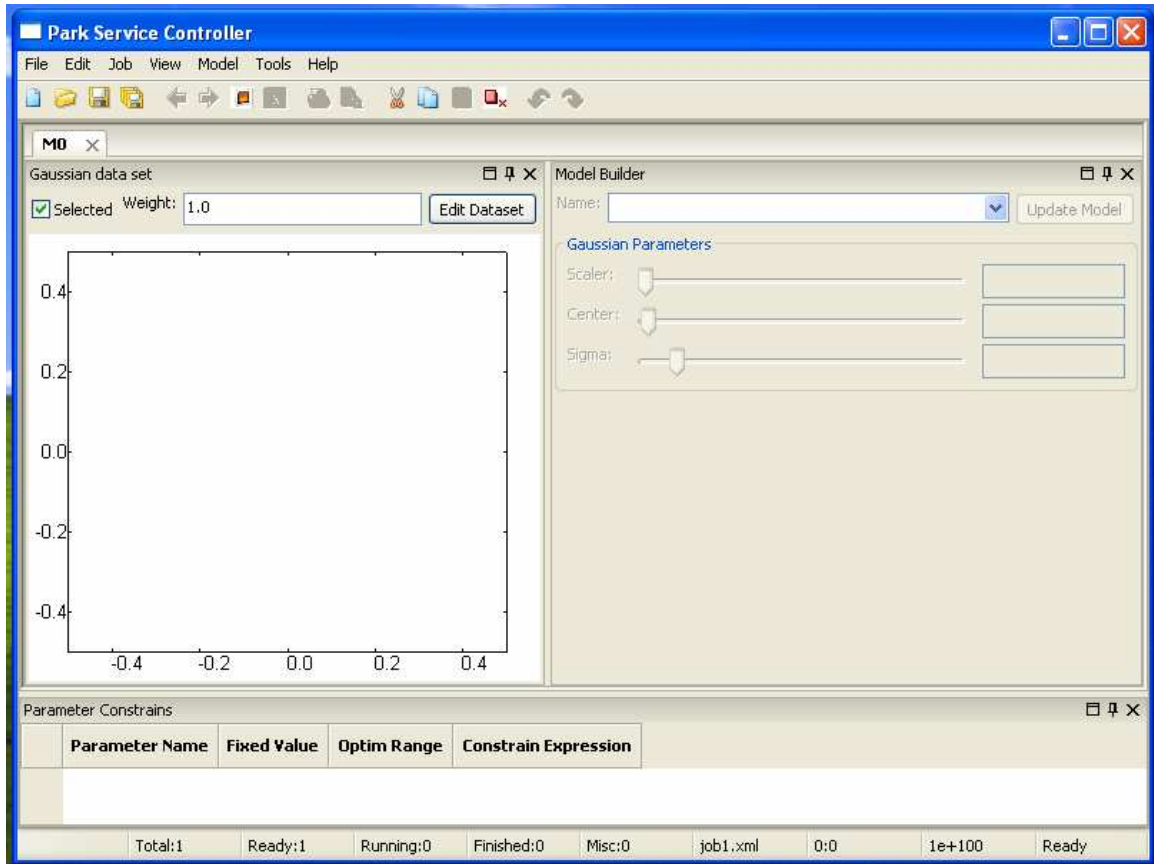
##### Residual Viewer #####
from fittingViewer1D import FittingViewer1D
#####

""" Register the model builder and its viewers. """
AVAILABLE_MODELS = ('gauss',)
## The registered models: name:(Menu_Item, Model_Page, Fitting_Viewer)
MODELS = {
    'gauss':('Gauss', GaussModelPage, FittingViewer1D),

```

}

With this configuration, the following figure will be when running \$PARK/parkClien.py:



The Gaussian model can be create by the menu item: **Model->Gauss**, the model page for this Gaussian model is shown as above, and is ready for the fitting.

## ***Summary***