Code Sample: Accelerating IBM ILOG CPLEX Optimization Studio by integrating into IBM Platform Symphony

Purpose

In this sample we show how to develop a distributed application that uses IBM ILOG CPLEX Optimization Studio ILOG Concert technology 12.6 to solve a linear programming problem in a distributed fashion using IBM Platform Symphony 6.1. Optimizing a linear programing problem requires significant computing power. IBM Platform Symphony is leveraged to distribute the individual compute tasks into a grid and thereby increase performance.

Background

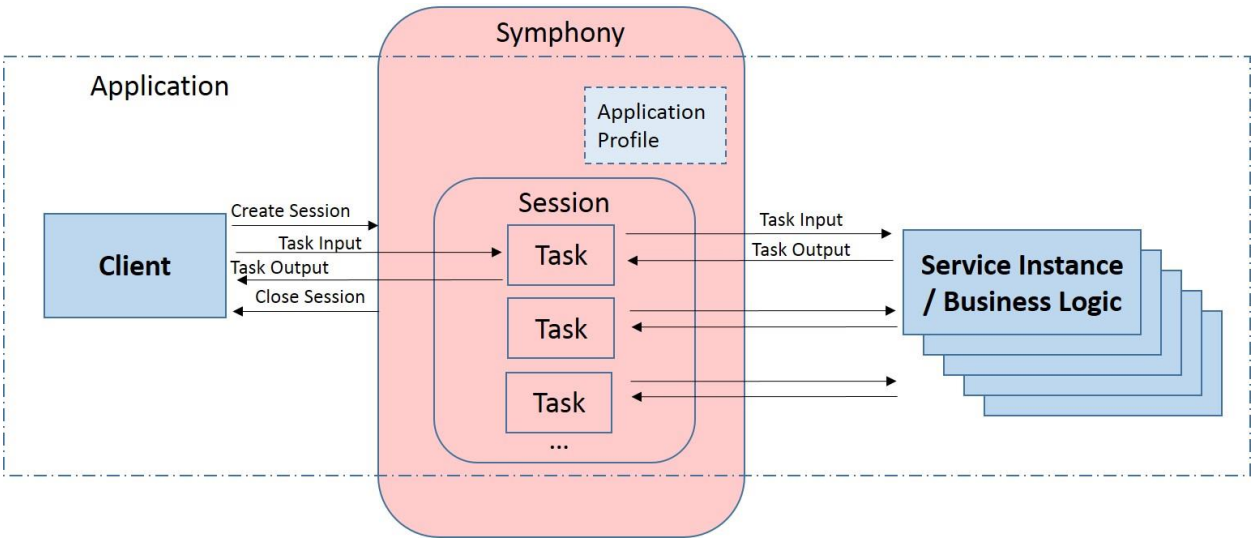
IBM Platform Symphony

IBM Platform Symphony provides powerful capabilities to run and manage distributed applications on a shared, scalable grid. It optimizes the utilization of available compute resources while improving and accelerating the applications achieving faster performance at reduced cost for infrastructure and maintenance.

Platform Symphony SOA applications consist of a client that connects to the management servers in Platform Symphony to submit requests, a service implementation that contains the business logic, and an application profile that describes the service and sets the necessary configuration parameters.

To submit a request, the client connects to Platform Symphony and creates a session. Tasks are then submitted to the session via an input message that also carries the data needed by the service to process the request.

The following picture illustrates these interactions at a high level:



Client and service are implemented using the Platform Symphony SDK, which provides the necessary APIs to interact with Platform Symphony. The SDK is available in a variety of programming languages. This example is using C++.

Developing or adapting applications for Platform Symphony is beyond the scope of this example. However, this is documented in great detail in the IBM Knowledge Center. Follow the link to the latest version of IBM Platform Symphony to find detailed background information, API references, and tutorials.

IBM ILOG CPLEX Optimization Studio

IBM ILOG CPLEX Optimization Studio is a set of components used to optimize business problems expressed in mathematical or linear programming models. It can solve a variety of programs, including linear programs, quadratic programs, and quadratically constraint programs, as well as their mixed integer variants. IBM ILOG CPLEX Optimization Studio is intended to be used by optimization experts that can translate complex business problems into models that can be solved programmatically using an optimization engine.

IBM ILOG CPLEX Optimization Studio is comprised of the following components:

- OPL: The optimization programming language is a scripting language used to develop the mathematical models
- An IDE to develop those models
- CPLEX Optimizer engine to solve the mathematical models
- CP Optimizer engine to solve mathematical models that require constraint programming techniques.

Together, these components allow optimization experts in cooperation with software developers to simply and efficiently develop applications to solve complex mathematical problems that are underlying real business decisions.

It is the purpose of this example to demonstrate how solving an optimization problem can be done even faster by using IBM Platform Symphony in conjunction with IBM ILOG CPLEX Optimization Studio.

The use case – portfolio optimization

A portfolio is a set of investments that is selected by an investor. Investors allocate a certain portion of their wealth to those investments and commonly have the following two goals:

1. To maximize the returns from the portfolio.
2. To reduce the uncertainty or variance for those returns (that is “risk”).

Risk reduction often comes at the price of a reduced return, so the optimal solution for a portfolio selection differs for each individual investor based on the degree to which they are willing to sacrifice return for certainty.

An analyst can make a reasonable assessment of securities in the market to determine their expected future returns and also the degree to which the returns are correlated. This data, combined with the level of risk aversion of the investor, leads to an optimal allocation of investment funds in the given securities.

Using some broad strokes and without getting into much detail about portfolio theory, we can state some objectives and constraints:

Assuming a set of investments I with returns r[i], and allocated funds a[i] our total return R is

(a) sum(i) a[i]*r[i]

We also know that there is a variance in the return of each investment and the investments are correlated, that is a change in return for one investment results in a change for other investments, as well. The deviation for each investment as well as a correlation coefficient are expressed in the covariance matrix C. It is beyond the scope of this example to describe this matrix further, but we can use this to define the portfolio return variance as

(b) (sum(i, j) a[i]*a[j] * C[i, j])/2

Using this as measure of risk and factoring in the investor's risk aversion *rho* (0<=rho<=1) we can state as objective:

(c) Determine max(sum(i) a[i]*r[i] – rho*(sum(i, j) a[i]*a[i]*C[i, j])/2),

in other words maximize return minus variance weighted by a measure of the investor's risk aversion.

A formal description and discussion of the model can be found in the paper at http://cowles.econ.yale.edu/P/cm/m16.

Sample code for the portfolio application

The example code that accompanies this document implements a portfolio optimization service. It illustrates how to write a service that uses IBM ILOG CPLEX Optimization Studio to solve optimization problems in IBM Platform Symphony.

The optimization service takes as input:

- A list of investments. For each investment an expected return is specified.
- A covariance matrix. The Covariance between two investments models the interdependency of these two investments.
- An initial wealth, the money we are allowed to invest.
- A risk factor rho in [0,1]. The optimization service balances variance vs. return. Small values of rho mean high risk, that is, higher return but with higher variance. Large values of rho mean less risk but also less return.

Based on this input the portfolio service computes an optimal portfolio allocation. It returns for each investment the optimal allocation. It also returns the expected return and variance of the optimal portfolio allocation.

To invoke our portfolio service a client will need to send a task input message through Platform Symphony and pass the data needed by the service. The task input message serves as input to the optimization function that will execute within the service. The result of the optimization function will be passed back to the client as a task output message.

To facilitate this communication, we first subclass Platform Symphony's Message class to implement our input and output messages, as can be seen in the following code snippets.

```
class Input : public soam::Message {
private:
   std::vector<Investment> mInvestments;
   Covariance mCovariance;
   double mWealth;
   double mRho;
. . .
```

Besides the data from the request the output message holds the result of the optimization, which is the total return and total variance for the optimized portfolio, as well as a vector of investment allocations that comprise the optimized portfolio itself:

```
class Output : public soam::Message {
   Output(Output const &);
   Output &operator=(Output const &);

   bool mOptimal;
   double mWealth;
   double mRho;
   double mObjValue;
   double mTotalReturn;
   double mTotalVariance;
   std::vector<Investment> mInvestments;
. . .
```

We implement an input message class for our client to submit requests, and an output message class that will hold the results coming from the service. We also define classes Investment, Covariance, etc. to hold the data needed by the service. This data will need to be serialized as part of the request and response transmission, so these classes will need to implement the serialization and deserialization of the data structures.

See the detailed definition for those classes and structs in the header files in the `Common` directory of the code example.

The portfolio service implementation

To create a service we need to extend Platform Symphony's *ServiceContainer* class. This class has methods that are called by the middleware when triggered by certain events, for example *onCreateService()* is called right after the service instance is started. We don't need to override any of the handler methods except for *onInvoke()*. The *onInvoke()* method is called each time a task input is sent to be processed by the service and therefore contains our business logic. This is the only method we override in our code example:

```
class Service : public ServiceContainer {
  public:
     virtual void onInvoke(TaskContextPtr& taskContext)
     {
        /*****************************************
         * Do your service logic here. This call
         * applies to each task submission.
         *****************************************/
. . .
```

Next we retrieve the data for the optimization from the task input message by calling *populateTaskInput()*. We can get the overall wealth, the covariance matrix, and the investor's risk aversion factor rho from the task input message.

```
. . .
   Input input;
   taskContext->populateTaskInput(input);
   Covariance const &covariance = input.getCovariance();
   wealth = input.getWealth();
   rho = input.getRho();
   investments = input.getInvestments();
. . .
```

With the provided input we can now use the CPLEX API to set up our model and our objective and then call the CPLEX optimizer to determine an optimal allocation of funds to each investment. How the optimization problem is set up using *IloEnv*, *IloModel*, and other CPLEX API classes can be seen in the example's source code, and you can find documentation for the API and IBM ILOG CPLEX Optimization Studio in the IBM Knowledge Center. The core of this example is also part of the IBM ILOG CPLEX Optimization Studio examples and is documented in greater detail there.

When the optimization problem is set up in CPLEX, we call the optimizer to solve and determine an optimal solution if one exists.

```
. . .
// Create a CPLEX instance and solve
// the optimization problem.
IloCplex cplex(env);
. . .
cplex.extract(model);
bool const feasible = cplex.solve();
. . .
```

The optimal solution is defined by a vector of allocations to investments along with the total expected return and the total variance for this portfolio. This optimal allocation is the solution to our objective and needs to be sent back to the client in an output message.

```
taskContext->setTaskOutput(output);
```

This is all we need for our Service class. The only thing remaining is to provide an entry point for the service in the *main()* method which starts the service binary:

```
int main(int argc, char* argv[])
{
   // Return value of our service program
   int returnValue = 0;

   try
   {
      // Create the container and run it
      cpx::portfolio::Service().run();
}
   catch(soam::SoamException& exp) {
      // Report the exception to stdout
      std::cout << "exception caught ... " << exp.what() << std::endl;
      returnValue = -1;
   }

return returnValue;
}
```

Note that the return value in the code section above is the return value of the binary and is not part of the return of a service request.

The complete code for this service is in the `PortfolioService.cpp` file in the `Service` directory of the code example.

The portfolio client implementation

Here we create a simple command line client that connects to Platform Symphony and submits the input data. The client code is contained in the `Client` directory of the code example in the file `PortfolioClient.cpp`.

Before we can interact with Platform Symphony through the client APIs we need to initialize the API. We also need to specify the name of the Symphony application we want to connect to, and provide credentials for the connection. Once the connection with Platform Symphony is established a session can be created, as in the following code snippet:

```
SoamFactory::initialize();

// Set up application specific information
char appName[] = "PortfolioClient";

// Set up application authentication information using
// the default security provider.
DefaultSecurityCallback securityCB("Guest", "Guest");

// Connect to the specified application.
ConnectionPtr conPtr = SoamFactory::connect(appName, &securityCB);

<. . .>

// Create a new session
SessionCreationAttributes attributes;
attributes.setSessionName("mySession");
attributes.setSessionType("ShortRunningTasks");
attributes.setSessionFlags(Session::ReceiveSync);

sesPtr = conPtr->createSession(attributes);
```

Now we can submit our requests to the portfolio service. To do that we create the task input message with the investment data required by the portfolio service. We consider the investment data and the total wealth of the investor a constant value, and then create different scenarios for varying risk aversion values rho. So each optimization problem will differ in the rho value only, which means we set up our *Input* with the investment and wealth data, and then iterate over all rho values to submit a different request for each value of rho.

```
// Prepare the input message for the service.
Input input;
. . .
input.setInvestments(investments);
input.setCovariance(covar);
input.setWealth(wealth);
input.setRho(rhomin);
. . .
// Submit the input message to start optimization.
TaskSubmissionAttributes attrTask;
TaskInputHandlePtr inputHandle = 0;

for (double d=rhomin; d <= rhomax; d+=step) {
   input.setRho(d);
   attrTask.setTaskInput(&input);
   inputHandle = sesPtr->sendTaskInput(attrTask);
   . . .
}
. . .
```

The last line in this code snippet, *sendTaskInput(attrTask)*, submits the task input to Platform Symphony, which will then forward it to the portfolio service and invoke the service. These requests can be processed in parallel by different instances of our service.

The client can now exit and come back later to retrieve the results, or it can wait. We can retrieve the results by calling *fetchTaskOutput()* and then iterate over the task results to get the output for each task by calling *populateTaskOutput():*

```
. . .

TaskOutputHandlePtr outputHandle;
. . .
// Get the message returned from the service.
Output output;
outputHandle->populateTaskOutput(&output);

. . .
```

After receiving an optimized portfolio, we need to close the session, disconnect from Platform Symphony, and uninitialized the API:

```
sesPtr->close();

< . . .>

/*******************************************************
 * It is important that we always uninitialize the API.
 * This is the only way to ensure proper shutdown of the
 * interaction between the client and the system.
 *******************************************************/
SoamFactory::uninitialize();
```

Java GUI client

There is also a Java Swing GUI client in the `java` directory of the code example that allows interactive editing and graphical display of results. The Java GUI client follows the same principals as the command line client.
The main component in the GUI is a table that allows specification of investments, returns, and covariance. The GUI offers two ways to use the portfolio optimization service:

1. Click **Run** to find the optimal portfolio allocation for the specified rho and wealth values. Multiple rho and/or wealth can be specified as a blank separated list of values. The GUI will create a separate task for each combination of those values. For each task the GUI will also create a new tab that displays the results.

2. Click **Sample** to sample the rho parameter over the specified interval with the specified step width. For each sampling value of rho the GUI will submit a separate task to the portfolio optimization service.
   Sampling results are displayed in a graph as they come in on a new tab.

Build the code

To build the sample code, we need to set up both, IBM ILOG CPLEX Optimization Studio and IBM Platform Symphony Developer Edition. Trial versions for both software bundles can be downloaded for

free from IBM's website. The versions we're using to compile the code samples are IBM ILOG CPLEX Optimization Studio 12.6.1 and IBM Platform Symphony Developer Edition 6.1. For the purpose of this document we were using a RHEL 6.4 system. Use equivalent steps for other operating systems.

Setting up the build environment takes just a few steps:

1. Download and unzip the attached sample code into a local directory. The contents should look as in the following picture:

```
[root@symphony9]# ls
Client  CplexPortfolioService.xml  java      Output  Service
Common  investments.dat            Makefile  README
[root@symphony9]#
```

2. Follow the instructions that come with the software bundles to install IBM ILOG CPLEX Optimization Studio and Platform Symphony Developer Edition.

3. Source the Platform Symphony Developer Edition profile applicable to your environment, for example profile.platform, to set environment parameters for Platform Symphony Developer Edition.

4. Modify the Makefile in the root directory to set SYMPHONY_HOME and COS_HOME to the installation directories for Platform Symphony Developer Edition and IBM ILOG CPLEX Optimization Studio, respectively. The compiler will use those to find libraries necessary for compilation. If needed, also change COS_SYSLIB to the system OS and architecture portion of the library path. You might need to check the IBM ILOG CPLEX Optimization Studio installation directory to find the correct setting. The compiler should be able to find the required libraries in `<COS_HOME>/cplex/lib/<COS_SYSLIB>`.

5. Run **make** to build the client and service executables and the service package. The service package is a tar.gz file that is used to deploy the application to Platform Symphony. Build artifacts and executables are created in the `Output` directory.

6. Note: The Java client which is also included in the sample code is not built by default. To build the Java GUI client you have to explicitly call **make gui**. This requires a Java 1.7 SDK to be installed and the JAVA_HOME variable set in the environment.

Before the application can be used, it has to be registered and deployed to Platform Symphony. You can do this using the Platform Symphony GUI console or using the CLI commands **soamdeploy** and **soamreg**. Details can be found in the IBM Knowledge Center. The application profile needed for the registration of the sample application can be found in the root directory of the code package.

Results

Once the application is registered we can run the client `PlatformClient` with its hard coded sample data, or pass in the enclosed data file. Run

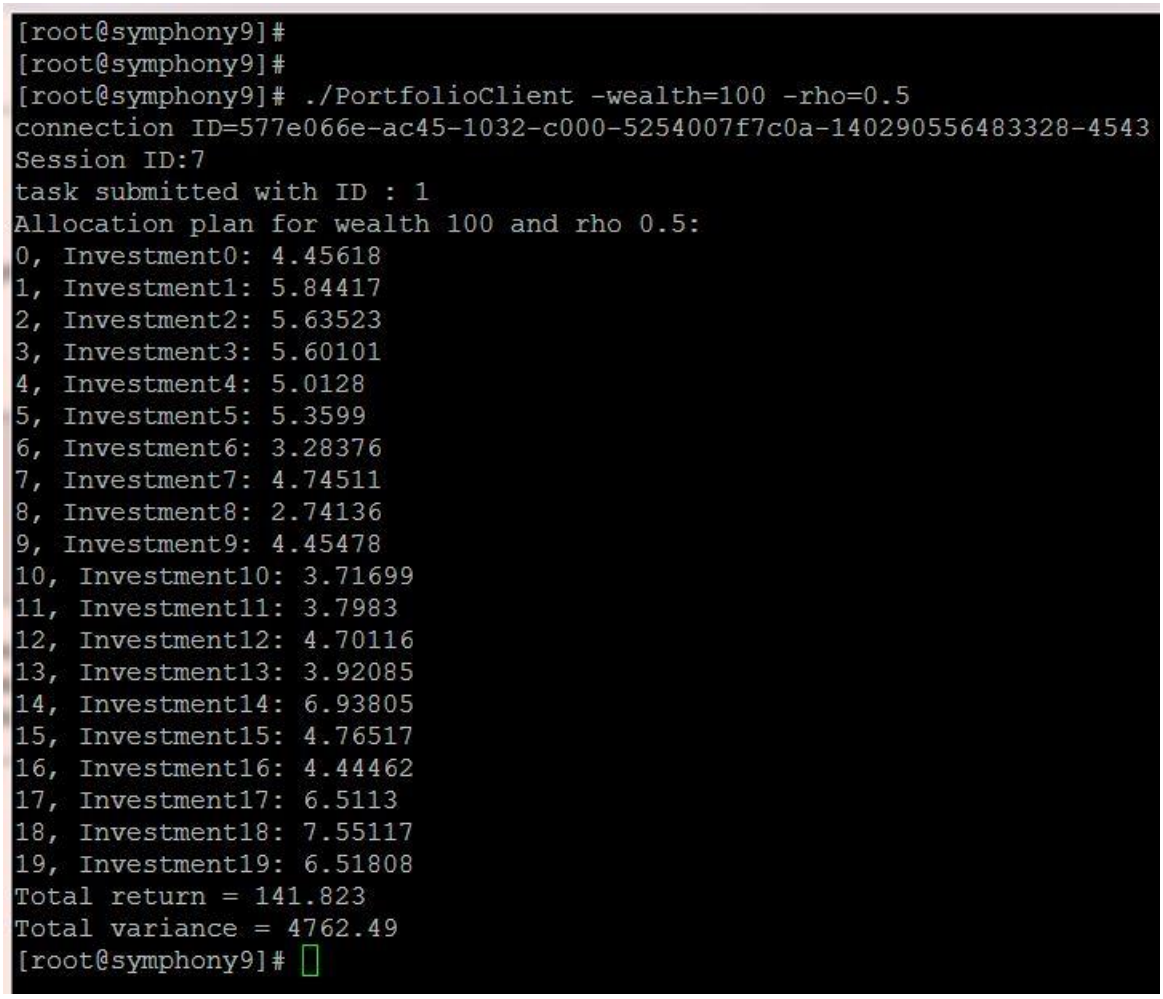
```
PortfolioClient -help
```

for more information about the command line options.

We need to specify a value for rho and the total wealth, for example:

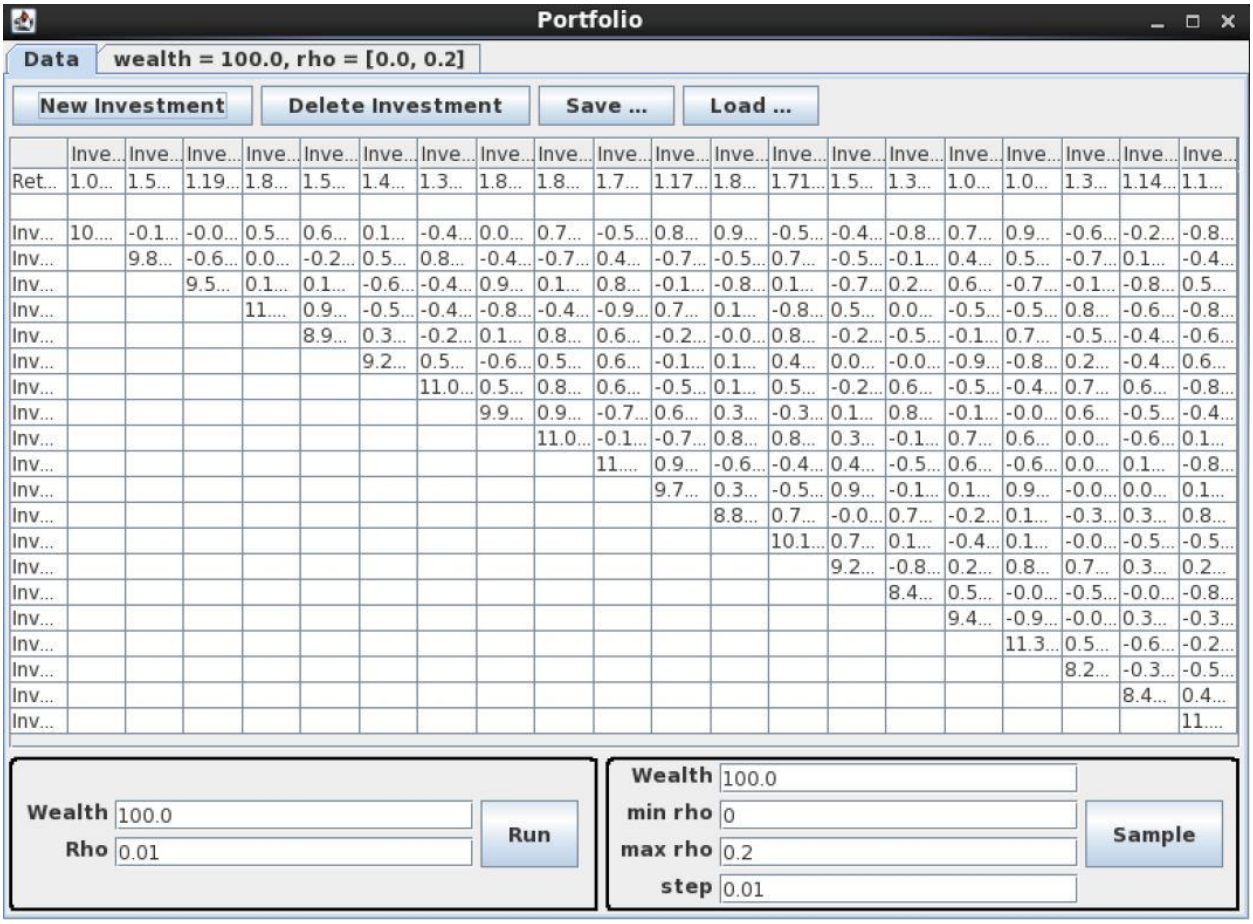
```
PortfolioClient –rho=0.5 –wealth=100
```

As can be seen in program output in the following picture, the client sets up a session with Platform Symphony, and submits the tasks:

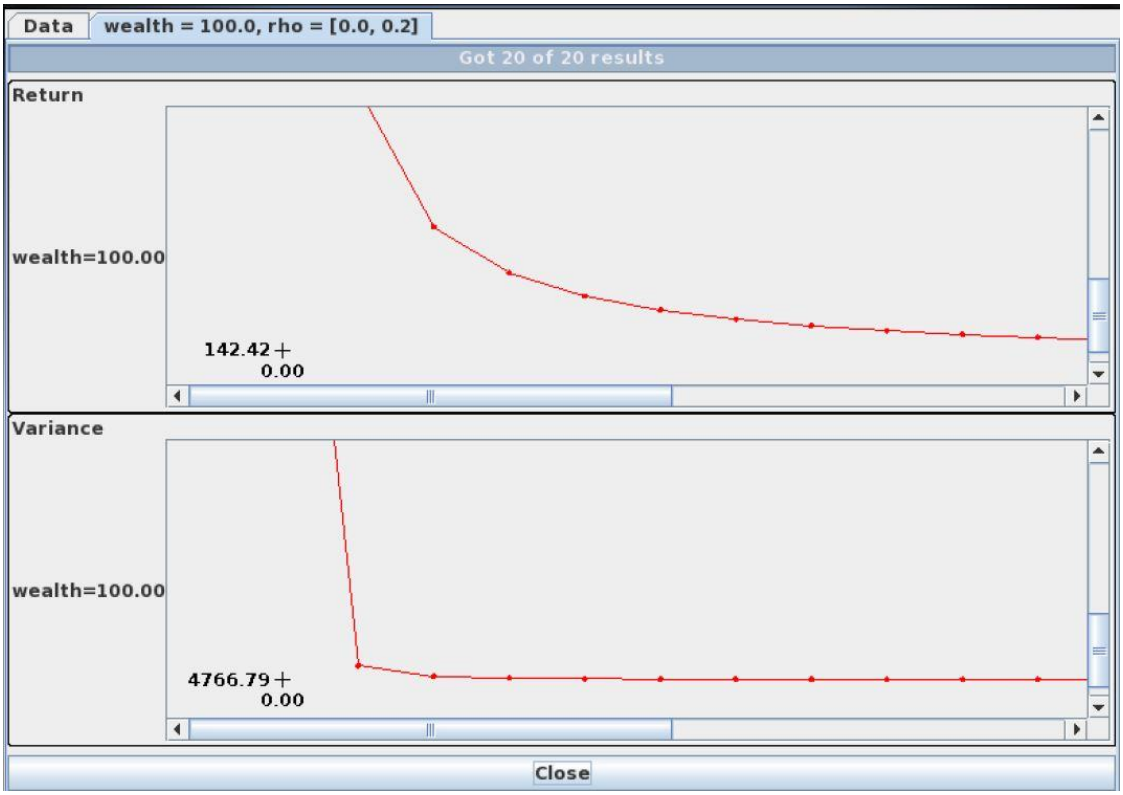


The result is an optimized set of allocations to the various investments, along with a total return and total variance.

To start the GUI client run **run.sh** in the `java` directory of the code package. This will start the Java GUI for this application. The GUI shows the investments and covariance in table form, as in the following screenshot:



The Java GUI client has the option of running a single rho value, or a range of values for rho over the sample data. Results are displayed in tabs along the top of the GUI client:



If a range of rho values is submitted, each value results in a separate task, and the results are displayed graphically over time as the results are received from Platform Symphony. The resulting diagrams show variance and return against values of rho

Software used in this example

IBM Platform Symphony

Trial download - IBM Platform Symphony Developer Edition

Trial download - IBM ILOG CPLEX Optimization Studio