GENERALIZED OPTIMAL TRADING TRAJECTORIES: A FINANCIAL QUANTUM COMPUTING APPLICATION

Marcos López de Prado †

This version: June 7, 2015

FIRST DRAFT – DO NOT CITE WITHOUT THE AUTHOR'S PERMISSION

† Senior Managing Director, Guggenheim Partners, New York, NY 10017. Research Fellow, Computational Research Division, Lawrence Berkeley National Laboratory, Berkeley, CA 94720. E-mail: lopezdeprado@lbl.gov, Website: www.QuantResearch.org

I would like to acknowledge useful comments from David H. Bailey (Lawrence Berkeley National Laboratory), José Blanco (Credit Suisse), Jonathan M. Borwein (University of Newcastle), Peter Carr (Morgan Stanley, NYU), Matthew D. Foreman (University of California, Irvine), Phil Goddard (1QBit), Andrew Landon (1QBit), Riccardo Rebonato (PIMCO, University of Oxford), Luis Viceira (HBS) and Jim Qiji Zhu (Western Michigan University).

GENERALIZED OPTIMAL TRADING TRAJECTORIES: A FINANCIAL QUANTUM COMPUTING APPLICATION

ABSTRACT

Generalized dynamic portfolio optimization problems have no known closed-form solution. These problems are particularly relevant to large asset managers, as the costs from excessive turnover and implementation shortfall may critically erode the profitability of their investment strategies.

In this brief note we demonstrate how this financial problem, intractable to modern supercomputers, can be reformulated as an integer optimization problem. Such representation makes it amenable to quantum computers.

Keywords: High-performance computing, integer optimization, quantum computing, adiabatic process.

JEL Classification: G0, G1, G2, G15, G24, E44.
AMS Classification: 91G10, 91G60, 91G70, 62C, 60E.

1. INTRODUCTION

A supercomputer is a mainframe computer able to perform an extremely large number of floating point operations per second (FLOPS). This is generally achieved following one of two approaches. In the first approach, a problem is divided into many small problems that can be solved in parallel. This is the strategy used by distributed computing or hyper-threaded architectures, such as cloud systems or GPUs. The second approach takes advantage of the topological configuration of a system to save time in I/O and other intensive operations. This is the key advantage of computer clusters. Moore's law, which states that the number of transistors on a chip will double approximately every two years, means that a system qualifies as a supercomputer for a relatively short period of time. The TOP500 project keeps track of the 500 fastest supercomputers in the world. As of June of 2014, 233 of these systems are located in the United States, and 76 in China.

Combinatorial optimization problems can be described as problems where there is a finite number of feasible solutions, which result from combining the discrete values of a finite number of variables. As the number of feasible combinations grows, an exhaustive search becomes impractical. The traveling salesman problem is an example of a combinatorial optimization problem that is known to be NP-hard, i.e. the category of problems that are at least as hard as the hardest problems solvable is nondeterministic polynomial time.

What makes an exhaustive search impractical is that standard computers evaluate and store the feasible solutions sequentially. But what if we could evaluate and store all feasible solutions at once? That is the goal of quantum computers. Whereas the bits of a standard computer can only adopt one of two possible states ($\{0,1\}$) at once, quantum computers rely on qubits, which are memory elements that may hold a *linear superposition* of both states. In theory, quantum computers can accomplish this thanks to quantum mechanics. A qubit can support currents flowing in two directions at once, hence providing the desired superposition. D-Wave, a commercial quantum computer designer, is planning to produce a 2048-qubit system in the year 2015. This linear superposition property is what makes quantum computers ideally suited for solving NP-hard combinatorial optimization problems.

In this note we will show how a dynamic portfolio optimization problem subject to generic transaction cost functions can be represented as a combinatorial optimization problem, tractable by quantum computers. Unlike Garleanu and Pedersen [2012], we will not assume that the returns are IID Normal. This problem is particularly relevant to large asset managers, as the costs from excessive turnover and implementation shortfall may critically erode the profitability of their investment strategies.

2. THE OBJECTIVE FUNCTION

Consider a set on assets $X = \{x_i\}$, $i = 1, \dots, N$, with returns following a multivariate Normal distribution at each time horizon $h = 1, \dots, H$, with varying mean and variance. We will assume that the returns are multivariate Normal, however not identically distributed through time. At a particular horizon h, we have a forecasted mean $\boldsymbol{\mu_h}$, a forecasted variance $\boldsymbol{V_h}$ and a forecasted transaction cost function $\tau_h[\boldsymbol{\omega}]$.

We define a trading trajectory as a *NxH* matrix $\boldsymbol{\omega}$ that determines the proportion of capital allocated to each of the *N* assets over each of the *H* horizons. This means that, given a trading trajectory $\boldsymbol{\omega}$, we can compute a vector of expected investment returns $\boldsymbol{r}$, as

$$\boldsymbol{r} = diag(\boldsymbol{\mu}^{\boldsymbol{T}}\boldsymbol{\omega}) - \tau[\boldsymbol{\omega}] \tag{1}$$

where $\tau[\boldsymbol{\omega}]$ can adopt any functional form. Without loss of generality, consider the following:

- $\tau_0[\boldsymbol{\omega}] = \sum_{n=1}^{N} c_{n,1}\sqrt{|\omega_{n,1} - \omega_n^*|}$.
- $\tau_h[\boldsymbol{\omega}] = \sum_{n=1}^{N} c_{n,h}\sqrt{|\omega_{n,h} - \omega_{n,h-1}|}$, for $h = 1, \dots, H$.
- ω_n^* is the initial holding of instrument $n, n = 1, \dots, N$.

$\tau[\boldsymbol{\omega}]$ is a *Hx1* vector of transaction costs. In words, the transaction costs associated with each asset is the sum of the square roots of the changes in capital allocations, re-scaled by an asset-specific factor $\{c_{n,h}\}$ that changes with *h*. Thus, a *Nx1* vector **C** determines the relative transaction cost across assets.

The Sharpe Ratio (SR) associated with $\boldsymbol{r}$ can be computed as ($\boldsymbol{\mu_h}$ being net of the risk-free rate)

$$SR[\boldsymbol{r}] = \frac{\sum_{h=1}^{H} \boldsymbol{\mu_h^T \omega_h}}{\sqrt{\sum_{h=1}^{H} \boldsymbol{\omega_h^T V_h \omega_h}}} \tag{2}$$

3. THE PROBLEM

We would like to compute the optimal trading trajectory that solves the problem

$$\begin{aligned} &\max_{\boldsymbol{\omega}} \; SR[\boldsymbol{r}] \\ &\text{s.t.}: \sum_{i=1}^{N} |\omega_{i,h}| = 1 \,, \forall h = 1, \dots, H \end{aligned} \tag{3}$$

Note that non-continuous transaction costs are embedded in $\boldsymbol{r}$. Compared to standard portfolio optimization applications, this is not a convex (quadratic) programming problem for at least three reasons: First, returns are not identically distributed, for $\boldsymbol{\mu_h}$ and $\boldsymbol{V_h}$ change with *h*. Second, transaction costs $\tau[\boldsymbol{\omega}]_h$ are non-continuous and changing with *h*. Third, the objective function $SR[\boldsymbol{r}]$ is not convex. Next, we will show how to calculate solutions without making use of any functional property of the objective function (hence the "generalized" nature of this approach).

4. AN INTEGER OPTIMIZATION APPROACH

The generality of this problem makes is intractable to standard convex optimization techniques. Our solution strategy is to discretize it so that it becomes amenable to integer optimization. This in turn allows us to use quantum computing technology to find the optimal solution.

4.1. PIGEONHOLE PARTITIONS

Suppose that we count with K units of capital, to be allocated among the N assets. This is a classic integer partitioning problem studied in number theory and combinatorics, and by Hardy and Ramanujan in particular, see Johansson [2012]. However, in our particular problem, order is relevant to the partition. For example, if $K=6$ and $N=3$, partitions $(1,2,3)$ and $(3,2,1)$ must be treated as different. This means that we must consider all distinct permutations of each partition (obviously $(2,2,2)$ does not need to be permutated). Next, we provide an efficient algorithm to generate the set of all partitions, $p^{K,N} = \left\{\{p_i\} | p_i \in \mathbb{Z}, \sum_{i=1}^{N} p_i = K\right\}$.

```
from itertools import combinations_with_replacement
#-----------------------------------------------------------------------
def pigeonHole(k,n):
   # Pigeonhole problem (organize k objects in n slots)
   for j in combinations_with_replacement(xrange(n),k):
      r=[0]*n
      for i in j:
         r[i]+=1
      yield r
```

Snippet 1 – Partitions of k objects into n slots

4.2. FEASIBLE STATIC SOLUTIONS

We would like to compute the set of all feasible solutions at any given horizon h, which we denote Ω. Consider a partition of K units into N assets, $p^{K,N}$. For each ith partition $\{p_i\}_{i=1,\ldots,N}$, we can define a vector of absolute weights such that $|\omega_i| = \frac{1}{K} p_i$, where $\sum_{i=1}^{N} |\omega_i| = 1$ (the full-investment constraint). This full-investment constraint implies that every weight can be either positive or negative, so for every vector of absolute weights $\{|\omega_i|\}_{i=1,\ldots,N}$ we can generate 2^N vectors of (signed) weights. This is accomplished by multiplying the items in $\{|\omega_i|\}_{i=1,\ldots,N}$ with the items of the Cartesian product of $\{-1,1\}$ with N repetitions. Snippet 2 shows how to generate the set Ω of all vectors of weights associated with all partitions, $\Omega = \left\{ \left\{ \frac{s}{K} p_i \right\} \middle| s \in \underbrace{\{-1,1\} \text{x} \ldots \text{x} \{-1,1\}}_{N}, p_i \in p^{K,N} \right\}$.

```
import numpy as np
from itertools import product
#-----------------------------------------------------------------------
def getAllWeights(k,n):
   #1) Generate partitions
   parts,w=pigeonHole(k,n),None
   #2) Go through partitions
   for part_ in parts:
      w_=np.array(part_)/float(k) #abs(weight) vector
      for prod_ in product([-1,1],repeat=n): #add sign
         w_signed_=(w_*prod_).reshape(-1,1)
         if w is None:w=w_signed_.copy()
```

```
        else:w=np.append(w,w_signed_,axis=1)
    return w
```

Snippet 2 – Set Ω of all vectors associated with all partitions

4.3. EVALUATING TRAJECTORIES

Given the set of all vectors Ω, we define the set of all possible trajectories Φ as the Cartesian product of Ω with H repetitions. That is, $\Phi = \{\{\Omega_h\}_{h=1,\ldots,H} | \Omega_h \in \Omega\}$. Then, for every trajectory we can evaluate its transaction costs, SR, and select the trajectory with optimal performance across Φ. Snippet 3 implements this functionality. The object **params** is a list of dictionaries that contain the values of $\boldsymbol{C}$, $\boldsymbol{\mu}$, $\boldsymbol{V}$.

```
import numpy as np
from itertools import product
#-------------------------------------------------------------------------
def evalTCosts(w,params):
    # Compute t-costs of a particular trajectory
    tcost=np.zeros(w.shape[1])
    w_=np.zeros(shape=w.shape[0])
    for i in range(tcost.shape[0]):
        c_=params[i]['c']
        tcost[i]=(c_*abs(w[:,i]-w_)**.5).sum()
        w_=w[:,i].copy()
    return tcost
#-------------------------------------------------------------------------
def evalSR(params,w,tcost):
    # Evaluate SR over multiple horizons
    mean,cov=0,0
    for h in range(w.shape[1]):
        params_=params[h]
        mean+=np.dot(w[:,h].T,params_['mean'])[0]-tcost[h]
        cov+=np.dot(w[:,h].T,np.dot(params_['cov'],w[:,h]))
    sr=mean/cov**.5
    return sr
#-------------------------------------------------------------------------
def dynOptPort(params,k=None):
    # Dynamic optimal portfolio
    #1) Generate partitions
    if k is None:k=params[0]['mean'].shape[0]
    n=params[0]['mean'].shape[0]
    w_all,sr=getAllWeights(k,n),None
    #2) Generate trajectories as cartesian products of weights with n repetitions
    for prod_ in product(w_all.T,repeat=len(params)):
        w_=np.array(prod_).T # concatenate product into a trajectory
        tcost_=evalTCosts(w_,params)
        sr_=evalSR(params,w_,tcost_) # evaluate trajectory
        if sr is None or sr<sr_: # store trajectory if better
            sr,w=sr_,w_.copy()
    return w
```

Snippet 3 – Evaluating all trajectories

Note that this procedure selects an optimally global trajectory without relying on convex optimization. A solution will be found even if the covariance matrices are ill-conditioned, transaction cost functions are non-continuous, etc. The price we pay for this generality is that calculating the solution is extremely computationally intensive. Indeed, evaluating all trajectories is a travelling-salesman problem. Digital computers are inadequate for this sort of NP-hard, however quantum computers have the advantage of evaluating multiple solutions at once, thanks to the property of linear superposition.

5. A NUMERICAL EXAMPLE

Here we illustrate how the global optimum can be found in practice, using a digital computer. A quantum computer would evaluate all trajectories at once, whereas the digital computer does this sequentially. First, we generate H vectors of means, covariance matrices and transaction cost factors, $\boldsymbol{C}$, $\boldsymbol{\mu}$, $\boldsymbol{V}$. These variables are stored in a **params** list.

```
import numpy as np
#---------------------------------------------------------------------------
def genMean(size):
    # Generate a random vector of means
    rMean=np.random.normal(size=(size,1))
    return rMean
#---------------------------------------------------------------------------
def genCovar(size):
    # Generate a random covariance matrix, as <A,A.T> of a random matrix
    rMat=np.random.rand(size,size)
    rCov=np.dot(rMat,rMat.T)
    return rCov
#---------------------------------------------------------------------------
    #1) Parameters
    size,horizon=3,2
    params=[]
    for h in range(horizon):
        mean_,cov_=genMean(size),genCovar(size)
        c_=np.random.uniform(size=cov_.shape[0])*np.diag(cov_)**.5
        params.append({'mean':mean_,'cov':cov_,'c':c_})
```

Snippet 4 – Generate the problem's parameters

Second, we can compute the performance of the trajectory that results from local (static) optima.

```
import numpy as np
#---------------------------------------------------------------------------
def statOptPortf(cov,a):
    # Static optimal porftolio
    # Solution to the "unconstrained" portfolio optimization problem
    cov_inv=np.linalg.inv(cov)
    w=np.dot(cov_inv,a)
    w/=np.dot(np.dot(a.T,cov_inv),a) # np.dot(w.T,a)==1
    w/=abs(w).sum() # re-scale for full investment
    return w
#---------------------------------------------------------------------------
```

```
#2) Static optimal portfolios
w_stat=None
for params_ in params:
   w_=statOptPortf(cov=params_['cov'],a=params_['mean'])
   if w_stat is None:w_stat=w_.copy()
   else:w_stat=np.append(w_stat,w_,axis=1)
tcost_stat=evalTCosts(w_stat,params)
sr_stat=evalSR(params,w_stat,tcost_stat)
print 'static SR:',sr_stat
```

Snippet 5 – Compute and evaluate the static solution

Third, we compute the performance associated with the global dynamic optimal trajectory.

```
import numpy as np
#-------------------------------------------------------------------
#3) Dynamic optimal portfolios
w_dyn=dynOptPort(params)
tcost_dyn=evalTCosts(w_dyn,params)
sr_dyn=evalSR(params,w_dyn,tcost_dyn)
print 'dynamic SR:',sr_dyn
```

Snippet 6 – Compute and evaluate the dynamic solution

The full code is listed in the appendix

6. REFERENCES

- Garleanu, N. and L. Pedersen (2012): "Dynamic Trading with Predictable Returns and Transaction Costs", Working paper.
- Johansson, F. (2012): "Efficient implementation of the Hardy-Ramanujan-Rademacher formula", LMS Journal of Computation and Mathematics 15, pp.341-359.

APPENDIX

A.1. INTEGER GLOBAL DYNAMIC PORTFOLIO OPTIMIZATION

This python 2.7 code implements the numerical example discussed in Section 5. The only dependencies are numpy and itertools.

```
# On 20150607 by MLdP <lopezdeprado@lbl.gov>
# Example for global dynamic optimization
import numpy as np
from itertools import combinations_with_replacement,product
#-------------------------------------------------------------------
def genMean(size):
   # Generate a random vector of means
   rMean=np.random.normal(size=(size,1))
   return rMean
#-------------------------------------------------------------------
def genCovar(size):
   # Generate a random covariance matrix, as <A,A.T> of a random matrix
   rMat=np.random.rand(size,size)
   rCov=np.dot(rMat,rMat.T)
   return rCov
#-------------------------------------------------------------------
def evalTCosts(w,params):
   # Compute t-costs of a particular trajectory
   tcost=np.zeros(w.shape[1])
   w_=np.zeros(shape=w.shape[0])
   for i in range(tcost.shape[0]):
      c_=params[i]['c']
      tcost[i]=(c_*abs(w[:,i]-w_)**.5).sum()
      w_=w[:,i].copy()
   return tcost
#-------------------------------------------------------------------
def evalSR(params,w,tcost):
   # Evaluate SR over multiple horizons
   mean,cov=0,0
   for h in range(w.shape[1]):
      params_=params[h]
      mean+=np.dot(w[:,h].T,params_['mean'])[0]-tcost[h]
      cov+=np.dot(w[:,h].T,np.dot(params_['cov'],w[:,h]))
   sr=mean/cov**.5
   return sr
#-------------------------------------------------------------------
def pigeonHole(k,n):
   # Pigeonhole problem (organize k objects in n slots)
   for j in combinations_with_replacement(xrange(n),k):
      r=[0]*n
      for i in j:
         r[i]+=1
      yield r
#-------------------------------------------------------------------
def statOptPortf(cov,a):
```

```
    # Static optimal porftolio
    # Solution to the "unconstrained" portfolio optimization problem
    cov_inv=np.linalg.inv(cov)
    w=np.dot(cov_inv,a)
    w/=np.dot(np.dot(a.T,cov_inv),a) # np.dot(w.T,a)==1
    w/=abs(w).sum() # re-scale for full investment
    return w
#-----------------------------------------------------------------------
def getAllWeights(k,n):
    #1) Generate partitions
    parts,w=pigeonHole(k,n),None
    #2) Go through partitions
    for part_ in parts:
        w_=np.array(part_)/float(k) # abs(weight) vector
        for prod_ in product([-1,1],repeat=n): # add sign
            w_signed_=(w_*prod_).reshape(-1,1)
            if w is None:w=w_signed_.copy()
            else:w=np.append(w,w_signed_,axis=1)
    return w
#-----------------------------------------------------------------------
def dynOptPort(params,k=None):
    # Dynamic optimal portfolio
    #1) Generate partitions
    if k is None:k=params[0]['mean'].shape[0]
    n=params[0]['mean'].shape[0]
    w_all,sr=getAllWeights(k,n),None
    #2) Generate trajectories as cartesian products of weights with n repetitions
    for prod_ in product(w_all.T,repeat=len(params)):
        w_=np.array(prod_).T # concatenate product into a trajectory
        tcost_=evalTCosts(w_,params)
        sr_=evalSR(params,w_,tcost_) # evaluate trajectory
        if sr is None or sr<sr_: # store trajectory if better
            sr,w=sr_,w_.copy()
    return w
#-----------------------------------------------------------------------
def main():
    #1) Parameters
    size,horizon=3,2
    params=[]
    for h in range(horizon):
        mean_,cov_=genMean(size),genCovar(size)
        c_=np.random.uniform(size=cov_.shape[0])*np.diag(cov_)**.5
        params.append({'mean':mean_,'cov':cov_,'c':c_})
    #2) Static optimal portfolios
    w_stat=None
    for params_ in params:
        w_=statOptPortf(cov=params_['cov'],a=params_['mean'])
        if w_stat is None:w_stat=w_.copy()
        else:w_stat=np.append(w_stat,w_,axis=1)
    tcost_stat=evalTCosts(w_stat,params)
    sr_stat=evalSR(params,w_stat,tcost_stat)
    print 'static SR:',sr_stat
```

```
    #3) Dynamic optimal portfolios
    w_dyn=dynOptPort(params)
    tcost_dyn=evalTCosts(w_dyn,params)
    sr_dyn=evalSR(params,w_dyn,tcost_dyn)
    print 'dynamic SR:',sr_dyn
    return
#-------------------------------------------------------------------------
if __name__=='__main__':main()
```

Snippet 7 – Full implementation of the integer global dynamic optimization problem