oBB Documentation

Release 0.1a

J. Fowkes

August 06, 2013

CONTENTS

oBB is an algorithm for the parallel global optimization of functions with Lipchitz continuous gradient or Hessian.

Warning: oBB is currently undergoing testing and may not work as expected.

CHAPTER

ONE

INSTALLING OBB

1.1 Requirements

oBB requires the following software to be installed:

- Python 2.6 to 2.7 (http://www.python.org/)
- A working implementation of MPI-2 (e.g. OpenMPI, http://www.open-mpi.org/)

Additionally, the following python packages should be installed (these will be installed automatically if using *pip*, see Installation using pip (recommended)):

- NumPy 1.3.0 or higher (http://www.numpy.org/)
- MPI for Python 1.3 or higher (http://mpi4py.scipy.org/)
- CVXOPT 1.1.3 or higher (http://cvxopt.org/)

Optionally, the following software may be manually installed for added functionality:

- matplotlib 1.1.0 or higher (http://www.matplotlib.org/) - for visualising the algorithm in 2D

1.2 Installation using pip (recommended)

For easy installation, use *pip* (http://www.pip-installer.org/):

```
$ [sudo] pip install obb
```

or alternatively *easy_install* (deprecated):

```
$ [sudo] easy_install obb
```

If you do not have root privileges or you want to install oBB for your private use, you can use:

```
$ pip install --user obb
```

which will install oBB in your home directory.

1.3 Manual installation

Alternatively, you can download the source code and unpack as follows:

```
$ wget http://pypi.python.org/packages/source/o/oBB/oBB-X.X.tar.gz
$ tar -xzvf oBB-X.X.tar.gz
$ cd oBB-X.X
```

and then build and install manually using:

```
$ python setup.py build
$ python setup.py install
```

If you do not have root privileges or you want to install oBB for your private use, you can use:

```
$ python setup.py install --user
```

instead.

1.4 Uninstallation

If oBB was installed using *pip* you can uninstall as follows:

```
$ [sudo] pip uninstall obb
```

If oBB was installed manually you have to remove the installed files by hand (located in your python site-packages directory).

CHAPTER

TWO

USER GUIDE

This section describes the main interface to oBB and how to use it.

2.1 Global Optimization

oBB is designed to solve the global optimization problem

$$\begin{aligned} &\min_{x \in \mathcal{R}^n} f(x) \\ \text{s.t. } &l \leq x \leq u \\ \text{and } &lc \leq Ax \leq uc \end{aligned}$$

where the domain $\mathcal{D}$ is convex, compact and the objective function f has Lipschitz continuous gradient or Hessian. oBB does not need to know the Lipschitz constants explicitly, it merely requires the user to supply elementwise bounds on the Hessian or derivative tensor of the objective function (see How to use oBB).

oBB uses local first or second order Taylor type approximations over balls within a parallel branch and bound framework. As with all branch and bound algorithms, the curse of dimensionality limits its use to low dimensional problems. The choice of whether to use first or second order approximations is down to the user (see How to use oBB).

For an in-depth technical description of the algorithm see the tech-report [CFG2013] and the paper [FGF2013].

2.2 How to use oBB

oBB requires the user to write a python script file which defines the functions and parameters necessary to solve the problem. These are determined by the choice of a first or second order approximation model (see [CFG2013] for details):

- First order models: *q* - norm based, *g*, *Hz*, *lbH*, *E0*, *Ediag* - minimum eigenvalue based
- Second order models: *c* - norm based, *gc* - minimum eigenvalue based

If using a first order model, the following functions are required:

- **f(x)** - returns objective function f at point x (scalar)
- **g(x)** - returns gradient $\nabla_x f$ of objective function at x (1D numpy array)

and the bounding function:

- **bndH(l,u)** - returns two numpy matrices of elementwise lower and upper bounds on the Hessian $\nabla_{xx} f$ of the objective function over $[l, u]$

For a second order model we also require:

- **H(x)** - returns Hessian $\nabla_{xx} f$ of objective function at x (2D numpy array)

and rather than bndH the bounding function:

- **bndT(l,u)** - returns two third order numpy tensors of elementwise lower and upper bounds on the derivative tensor $\nabla_{xxx} f$ of the objective function over $[l, u]$

We now need only specify the type of parallel branch and bound algorithm to use (again, see [CFG2013] for details):

- *T1* - bounds in parallel
- *T2_individual*, *T2_synchronised* - tree in parallel

See Example of Use for an in-depth worked example in python.

2.3 Optional Parameters

oBB allows the user to specify several optional parameters to control the behaviour of the algorithm:

- *tol* - objective function tolerance (e.g. 1e-2, the default)
- *toltype* - tolerance type (r - relative [default], a - absolute)
- *countf* - count objective function evaluations (0 - off, 1 - on [default])
- *countsp* - count subproblem evaluations (0 - off, 1 - on [default])

and if matplotlib (http://www.matplotlib.org/) is installed:

- vis - visualisation of the algorithm in 2D (0 - off [default], 1 - on)

Note that the linear constraint parameters A, lc and uc are also optional as the optimization problem is only required to be bound constrained.

2.4 Example of Use

Suppose we wish to solve the following problem:

$$\min_{x \in \mathcal{R}^n} \sum_{i=1}^{n} \sin(x_i)$$
$$\text{s.t.} \quad -1 \leq x_i \leq 1 \;\; \forall i = 1, \ldots, n$$
$$\text{and} \quad -1 \leq \sum_{i=1}^{n} x_i \leq 1$$

One can see that the gradient g, Hessian H and third order derivative tensor T are given by

$$g(x) = (\cos(x_1), \ldots, \cos(x_n))^T$$
$$H(x) = diag(-\sin(x_1), \ldots, -\sin(x_n))$$
$$T(x) = diagt(-\cos(x_1), \ldots, -\cos(x_n))$$

where $diagt$ is the tensor diagonal function (i.e. $diagt(v)$ places the vector v on the diagonal of the tensor).

It is straightforward to obtain elementwise bounds on the Hessian matrix H and third order derivative tensor T as both sin and cos can be bounded below and above by -1 and 1 respectively.

We can code this up in a python script file, let's call it sins.py as follows:

```
# Example code for oBB
from obb import obb
from numpy import sin,cos,diag,ones,zeros

# Input Settings
# Algorithm (T1, T2_individual, T2_synchronised)
alg = 'T2_synchronised'

# Model type (q - norm quadratic, g/Hz/lbH/E0/Ediag - min eig. quadratic,
# c - norm cubic, gc - gershgorin cubic)
mod = 'c'

# Tolerance
tol = 1e-2

# Tensor diagonal function
def diagt(v):
    T = zeros((D,D,D))
    for i in range(0,D):
            T[i,i,i] = v[i]
    return T

# Set up sum of sins test function
# Dimension
D = 2
# Constraints
l = -1*ones(D)
u = 1*ones(D)
```

```
A = ones((1,D))
lc = -1; uc = 1
# Required functions
f = lambda x: sum(sin(x))
g = lambda x: cos(x)
H = lambda x: diag(-sin(x))
bndH = lambda l,u: (diag(-ones(D)), diag(ones(D)))
bndT = lambda l,u: (diagt(-ones(D)), diagt(ones(D)))

# Name objective function
f.__name__ = 'Sum of Sins'

# Run oBB
xs, fxs, tol, itr = obb(f, g, H, bndH, bndT, l, u, alg, mod, A=A, lc=lc, uc=uc, tol=tc
```

This file is included in oBB as sins.py.

2.5 Running the Algorithm

To run the user-created python script file (e.g. sins.py) we need to execute it using MPI's mpiexec command, specifying the number of processor cores with the -n option. For example, to run oBB on four processor cores we simply execute the following shell command:

```
$ mpiexec -n 4 python sins.py
```

Note that if using the MPICH2 implementation of MPI we first need to start an mpd daemon in the background:

```
$ mpd &
```

but this is not necessary for other MPI implmentations, e.g. OpenMPI.

And that's all there is to it! Well, almost...

2.6 Using the RBF Layer

oBB can optionally approximate the objective function f by a radial basis function (RBF) surrogate and optimize the approximation instead (see [FGF2013] for details). The advantage of this approach is that the user merely needs to supply the objective function and a set of points at which the objective function should be evaluated to construct the RBF approximation. The disadvantage is that the optimum found by the algorithm will only be close to the optimum of the objective function if it is sampled at sufficiently many points.

As before, the user is required to write a python script file which defines the functions and parameters necessary to solve the problem. In addition to the approximation model, algorithm type parameters and objective function described in How to use oBB only an n by m numpy array of m points at which to sample the objective function needs to be specified.

See the RBF Layer Example below for an in-depth worked example in python.

2.7 RBF Layer Example

Suppose we wish to solve an RBF approximation to the problem give in the Example of Use section:

$$\begin{aligned} &\min_{x\in\mathcal{R}^n} \sum_{i=1}^{n} \sin(x_i) \\ \text{s.t.} \quad &-1 \le x_i \le 1 \;\; \forall i = 1,\ldots,n \\ \text{and} \quad &-1 \le \sum_{i=1}^{n} x_i \le 1 \end{aligned}$$

We can code this up in a python script file, let's call it sins_rbf.py as follows:

```
# Example RBF Layer code for oBB
from obb import obb_rbf
from numpy import sin,ones,zeros
from numpy.random import rand,seed

# Input Settings
# Algorithm (T1, T2_individual, T2_synchronised)
alg = 'T1'

# Model type (q - norm quadratic, g/Hz/lbH/E0/Ediag - min eig. quadratic,
# c - norm cubic, gc - gershgorin cubic)
mod = 'c'

# Tolerance
tol = 1e-2

# Set up sum of sins test function
# Dimension
D = 2
# Constraints
l = -1*ones(D)
u = 1*ones(D)
A = ones((1,D))
lc = -1; uc = 1
# Required functions
f = lambda x: sum(sin(x))

# Generate 10*D sample points for RBF approximation
seed(5) # !!Sample points have to be the same on all processors!!
pts = rand(10*D, D)

# Scale points so they lie in [l,u]
for i in range(0,D):
    pts[:,i] = l[i] + (u[i]-l[i])*pts[:,i]

# Name objective function
f.__name__ = 'RBF Sum of Sins'
```

```
# Run oBB
xs, fxs, tol, itr = obb_rbf(f, pts, l, u, alg, mod, A=A, lc=lc, uc=uc, tol=tol)
```

Note the use of obb_rbf instead of obb and the need for a random number seed so that the sample points are the same on all processors. This file is included in oBB as sins_rbf.py, to run it see Running the Algorithm.

2.8 RBF Layer for the COCONUT Test Set

oBB comes with a set of pre-computed RBF approximations to selected functions from the COCONUT test set that were used to produce the numerical results in the paper [CFG2013]. In order to optimze these approximations using oBB, the user is required to write a python script file which defines the desired function and tolerance (see [CFG2013] for a list of all 31 functions available). For example, to optimize an RBF approximation to the 'hs041' function the user could write the following python script file, let's call it coconut.py:

```
# Example COCONUT RBF code for oBB
from obb import obb_rbf_coconut

# Input Settings
# Algorithm (T1, T2_individual, T2_synchronised)
alg = 'T1'

# Model type (q - norm quadratic, g/Hz/lbH/E0/Ediag - min eig. quadratic,
# c - norm cubic, gc - gershgorin cubic)
mod = 'c'

# Tolerance (positive number e.g. 1e-2 or '12hr')
tol = '12hr'

# Choose RBF approximation from COCONUT test
f = 'hs041'

# Run oBB
xs, fxs, tol, itr = obb_rbf_coconut(f, alg, mod, tol=tol)
```

Note the use of obb_rbf_coconut and the optional 12hr tolerance setting which runs the algorithm to the absolute tolerance obtained by a serial code in twelve hours (see [CFG2013] for details). Of course one can also specific a desired tolerance (e.g. 1e-2) as before. This file is included in oBB as coconut.py, to run it see Running the Algorithm.

2.9 References

BIBLIOGRAPHY

[CFG2013] Cartis, C., Fowkes, J. M. and Gould, N. I. M. (2013) 'Branching and Bounding Improvements for Global Optimization Algorithms with Lipschitz Continuity Properties', *ERGO Technical Report*, no. 13-010, pp. 1-33. http://www.maths.ed.ac.uk/ERGO/pubs/ERGO-13-010.html

[FGF2013] Fowkes, J. M. , Gould, N. I. M. and Farmer, C. L. (2013) 'A Branch and Bound Algorithm for the Global Optimization of Hessian Lipschitz Continuous Functions', *Journal of Global Optimization*, vol. 56, no. 4, pp. 1791-1815. http://dx.doi.org/10.1007/s10898-012-9937-9