pypgapack Documentation

Release 0.1.0

Jeremy Roberts

December 04, 2011

CONTENTS

CHAPTER

ONE

GETTING STARTED WITH PYPGAPACK

1.1 Background

`pypgapack` is a Python wrapper for the parallel genetic algorithm library pgapack, written in C by David Levine. The source and documentation for pgapack can be found at http://ftp.mcs.anl.gov/pub/pgapack/. The motivation for wrapping the code is ultimately to support a class project aiming to optimize loading patterns of nuclear reactor cores, which is a rather large and difficult combinatorial problem. Lots of researchers have applied genetic algorithms (and many other algorithms) to the problem, and the class project aims to provide a flexible test bench in Python to investigate various ideas. Wrapping pgapack is one step toward that goal. pgapAck was chose largely due to limited but positive past experience with it.

It should be pointed out that a similar effort to wrap pgapack in Python was made called `pgapy` (see http://pgapy.sourceforge.net/), but I actually couldn't get it to work, probably because I didn't know a thing about building Python modules before I started this (and my minimal C knowledge didn't help matters). Hence, I decided to "roll my own" using SWIG in combination with a C++ wrapper around pgapack instead of interfacing directly with pgapack as `pgapy` does.

The `PGA` class wraps almost all of pgapack's functionality, including allowing user functions for several operations (like initialization, crossover, etc.) for the `PGA.DATATYPE_BINARY`, `PGA.DATATYPE_REAL` and `PGA.DATATYPE_INTEGER` alleles. No such support is currently offered for other allele types, including user-specified types. The intended way to use `pypgapack` is to derive classes from `PGA`, with objective and other functions as members.

Parallel functionality is supported with the help of mpi4py.

Note: `pypgapack` is currently in beta mode, so there may be many things that look wrapped but are not. Testing is a future goal, but not a priority—I need a grade! Feedback is welcome at robertsj@mit.edu.

1.2 Building pypgapack

Included in ./pypgapack are the required source files and a simple script `build_pypgapack` which generates the Python module. To build, do the following:

1. Build PGAPack with the patches in ./patches. The major difference is a slight change to allow use with C++. The Makefile template also is set to produce shared and static libraries.

2. Modify the paths and variables in `build_pypgapack` below to suit your needs.
3. The source as distributed is set for serial. To use in parallel, do the following:
   - Uncomment `PARALLEL` in `build_pypgapack`
   - Set `CXX` to the appropriate compiler (e.g. mpic++) in `build_pypgapack`
   - Delete or move the dummy `mpi.h` included with PGAPack to avoid redefinitions. There's probably a better approach.
   - This assumes PGAPack was built in parallel; if not, do so. Refer to the PGAPack documentation. You need an MPI-enabled compiler.
   - Get mpi4py (e.g. easy_install mpi4py). You need an MPI-enabled compiler. Note, a few files from mpi4py are included in ./pypgapack/mpi4py. These *may* need to be updated.
4. Execute `build_pypgapack` and set PYTHONPATH accordingly.

1.3 Next Steps

The user is encouraged to read the pgapack documentation thoroughly before using pypgapack, as the shared API is *not* covered in this documentation (and neither are the many PGAPack defaults). It's helpful to go through their examples in C/C++ or Fortran if you know the languages.

Thereafter, see the collection of *Examples*, which include several of the original pgapack examples along with a few additional ones that demonstrate how to use user-defined functions for a variety of operations. Reference output is included, though don't expect to reproduce the numbers exactly for the small number of generations used, as they'll be sensitive to compilation, etc.

For a quick refresher, the basic gist of genetic algorithms is discussed briefly in *Methods*, which lists a few references that may be of use.

Documentation for the relatively small number of additional methods not explicitly in pgapack can be found in the *API Reference*.

CHAPTER

TWO

EXAMPLES

All examples are located in the `pypgapack/examples` and the reference output for all examples is in `pypgapack/examples/output`. Aside from small floating point differences, the values should be the same given the use of a fixed random number generator seed in all the examples. A utility script `run_examples.py` is included to test user output to the included reference cases. (Note, the above might actually be untrue, as compilation can and will change how a fixed pseudo-random number sequence is generated.)

Also, the maximum generation count is limited to 50 for all cases to produce short output. Experiment with that limit to see better solutions.

2.1 Basic Examples

The following are some simple examples that illustrate the basic PGAPack functionality.

2.1.1 Example 1: MAXBIT

`pypgapack` is pretty easy to use, and to demonstrate, we'll solve the maxbit problem, the first example in the PGAPack documentation.

```python
"""
pypgapack/examples/example01.py  --  maxbit
"""
from pypgapack import PGA
import sys
class MyPGA(PGA) :
    """
    Derive our own class from PGA.
    """
    def maxbit(self, p, pop) :
        """
        Maximum when all alleles are 1's, and that maximum is n.
        """
        val = 0
        # Size of the problem
        n = self.GetStringLength()
```

```
        for i in range(0, n) :
            # Check whether ith allele in string p is 1
            if self.GetBinaryAllele(p, pop, i) :
                val = val + 1
        # Remember that fitness evaluations must return a float
        return float(val)

# (Command line arguments, 1's and 0's, string length, and maximize it)
opt = MyPGA(sys.argv, PGA.DATATYPE_BINARY, 100, PGA.MAXIMIZE)
opt.SetRandomSeed(1)         # Set random seed for verification.
opt.SetMaxGAIterValue(50)    # 50 generations (default 1000) for short output.
opt.SetUp()                  # Internal allocations, etc.
opt.Run(opt.maxbit)          # Set the objective.
opt.Destroy()                # Clean up PGAPack internals
```

Running it yields the following output:

```
***Constructing PGA***
Iter #     Field      Value
10         Best       6.900000e+01
Iter #     Field      Value
20         Best       7.200000e+01
Iter #     Field      Value
30         Best       7.600000e+01
Iter #     Field      Value
40         Best       7.700000e+01
Iter #     Field      Value
50         Best       8.000000e+01
The Best Evaluation: 8.000000e+01.
The Best String:
[ 0111110111101111011111100111011110111101111111011111110100111111 ]
[ 011111110111110111111110111101011011 ]
***Destroying PGA context***
```

2.1.2 Example 2: MAXINT

This is a similar problem, but the alleles are integers ranging from -100 to 100. Note that when the integer ranges are set, a cast to `intc` is used. Python uses high precision datatypes, and there doesn't seem to be a safe implicit conversion between the Python integer type and the C integer type behind the scenes (in SWIG land). Casting explicitly circumvents the issue.

```
"""
pypgapack/examples/example02.py  --  maxint
"""
from pypgapack import PGA
import numpy as np
import sys
class MyPGA(PGA) :
    """
    Derive our own class from PGA.
    """
    def maxint(self, p, pop) :
```

```python
        """
        The maximum integer sum problem.

        The alleles are integers, and we solve
            max f(x) = x_1 + x_2 + ... + x_N
        subject to
            |x_i| <= 100 .
        That maximum is f(x) = 100n obtained for x_i = 100 for all i.
        """
        c = self.GetIntegerChromosome(p, pop) # Get pth string as Numpy array
        val = np.sum(c)                       #   and sum it up.
        del c                                 # Delete "view" to internals.
        return float(val)                     # Always return a float.

n   = 10                # String length.
# (Command line arguments, integers, string length, and maximize it)
opt = MyPGA(sys.argv, PGA.DATATYPE_INTEGER, n, PGA.MAXIMIZE)
opt.SetRandomSeed(1)    # Set random seed for verification.
u_b =  100*np.ones(n)   # Define lower bound.
l_b = -100*np.ones(n)   # Define upper bound.
# Set the bounds.  Note, need to cast as C-combatible integers.
opt.SetIntegerInitRange(l_b.astype('intc'), u_b.astype('intc'))
opt.SetMaxGAIterValue(50)    # 50 generations for short output.
opt.SetUp()                  # Internal allocations, etc.
opt.Run(opt.maxint)          # Set the objective.
opt.Destroy()                # Clean up PGAPack internals.
```

Running it yields the following output:

```
***Constructing PGA***
Iter #     Field      Value
10         Best       5.100000e+02
Iter #     Field      Value
20         Best       6.480000e+02
Iter #     Field      Value
30         Best       7.150000e+02
Iter #     Field      Value
40         Best       7.760000e+02
Iter #     Field      Value
50         Best       8.220000e+02
The Best Evaluation: 8.220000e+02.
The Best String:
#    0: [       27], [      100], [       86], [       98], [       97], [       99]
#    6: [       79], [       89], [       64], [       83]

***Destroying PGA context***
```

2.1.3 Example 3: MAXREAL

This is the same problem, but for real alleles. Here, note use of `SetMutationalType()` with the option of `PGA.MUTATION_RANGE`. This forces mutated allele values to remain within the initial range specified, useful for cases with constrained inputs. The default adds some (small) random amount, but over many iterations, this can cause allele values to go significantly beyond the initial range.

```
"""
pypgapack/examples/example03.py  --  maxreal
"""
from pypgapack import PGA
import numpy as np
import sys

class MyPGA(PGA) :
    """
    Derive our own class from PGA.
    """
    def maxreal(self, p, pop) :
        """
        The maximum real sum problem.

        The alleles are doubles, and we solve
        .. math::
          \max f(x) &= \sum^N_{n=1} x_n \\
              s.t.  &= |x_i| \leq 100
        That maximum is  :math:`f_{\textrm{max}}(x) = 100n` obtained for
        :math:`x_i = 100, i = 1\ldots N`.
        """
        c = self.GetRealChromosome(p, pop) # Get pth string as Numpy array
        val = np.sum(c)                    #   and sum it up.
        del c                              # Delete "view" to internals.
        return val                         # Already a float.

n   = 10                 # String length.
# (Command line arguments, doubles, string length, and maximize it)
opt = MyPGA(sys.argv, PGA.DATATYPE_REAL, n, PGA.MAXIMIZE)
opt.SetRandomSeed(1)   # Set random seed for verification.
u_b = 100*np.ones(n)   # Define lower bound.
l_b =-100*np.ones(n)   # Define upper bound.
# Set the bounds.  Default floats are handled without issue.
opt.SetRealInitRange(l_b, u_b)
# Force mutations to keep values in the initial range, a useful
#   feature for bound constraints.
opt.SetMutationType(PGA.MUTATION_RANGE)
opt.SetMaxGAIterValue(50) # 50 generations for short output.
opt.SetUp()               # Internal allocations, etc.
opt.Run(opt.maxreal)      # Set the objective.
opt.Destroy()             # Clean up PGAPack internals.
```

Running it yields the following output:

```
***Constructing PGA***
Iter #      Field       Value
10          Best        5.535524e+02
Iter #      Field       Value
20          Best        6.550377e+02
Iter #      Field       Value
30          Best        7.378405e+02
```

```
Iter #     Field      Value
40         Best       7.537097e+02
Iter #     Field      Value
50         Best       7.827324e+02
The Best Evaluation: 7.827324e+02.
The Best String:
#   0: [   66.95374], [   47.92554], [    91.9023], [   75.87186], [   96.31829]
#   5: [   85.24209], [   69.58556], [   90.59017], [   86.26947], [   72.07333]

***Destroying PGA context***
```

2.2 Examples of User Defined Operators

These examples explore one of the strengths of PGAPack, namely user-defined operators.

2.2.1 Example 4: User-defined String Initialization

We redo *Example 2: MAXINT* by initializing the strings with our own routine. Here, that's done by generating a permutation using Numpy.

```python
"""
pypgapack/examples/example04.py  --  maxint with user initialization
"""
from pypgapack import PGA
import numpy as np
import sys
class MyPGA(PGA) :
    """
    Derive our own class from PGA.
    """
    def maxint(self, p, pop) :
        """
        The maximum integer sum problem.
        """
        c = self.GetIntegerChromosome(p, pop) # Get pth string as Numpy array
        val = np.sum(c)                       #   and sum it up.
        del c                                 # Delete "view" to internals.
        return float(val)                     # Always return a float.

    def init(self, p, pop) :
        """
        Random permutations using Numpy.
        """
        n    = self.GetStringLength()
        c    = self.GetIntegerChromosome(p, pop)
        perm = np.random.permutation(n)
        for i in range(0, n) :
            c[i] = perm[i]
        del c

```

```python
n   = 10                 # String length.
# (Command line arguments, integers, string length, and maximize it)
opt = MyPGA(sys.argv, PGA.DATATYPE_INTEGER, n, PGA.MAXIMIZE)
opt.SetRandomSeed(1)     # Set random seed for verification.
np.random.seed(1)        # Do the same with Numpy.
u_b =  100*np.ones(n)    # Define lower bound.
l_b = -100*np.ones(n)    # Define upper bound.
# Set the bounds.  Note, need to cast as C-compatible integers.
opt.SetIntegerInitRange(l_b.astype('intc'), u_b.astype('intc'))
opt.SetMaxGAIterValue(50)     # 50 generations for short output.
opt.SetUp()                   # Internal allocations, etc.
opt.Run(opt.maxint)           # Set the objective.
opt.Destroy()                 # Clean up PGAPack internals.
```

Running it yields the following output:

```
***Constructing PGA***
Iter #     Field      Value
10         Best       5.100000e+02
Iter #     Field      Value
20         Best       6.480000e+02
Iter #     Field      Value
30         Best       7.150000e+02
Iter #     Field      Value
40         Best       7.760000e+02
Iter #     Field      Value
50         Best       8.220000e+02
The Best Evaluation: 8.220000e+02.
The Best String:
#    0: [       27], [      100], [       86], [       98], [       97], [       99]
#    6: [       79], [       89], [       64], [       83]

***Destroying PGA context***
```

2.2.2 Example 5: User-defined Crossover Operator

This example solves "Oliver's 30-city Hamiltonian cycle Traveling Salesman Problem", as described in Poon and Carter, *Computer Ops Res.*, **22**, (1995). More importantly, it demonstrates use of a user-defined crossover operator, namely the "Tie-Breaking Crossover" (TBX1) of the same work.

The problem has 30 cities in a plane, and the goal is to minimize the distance traveled when visiting each city just once in a complete loop. The problem has 40 equivalent optima, each with a distance of 423.741 units. The coordinates of the cities are in the code, and are from Oliver's original paper by way of Steve Dower's site. See *Parallel Examples* for a parallel version that tries matching the cited results.

```python
"""
pypgapack/examples/example05.py  --  traveling salesman
"""
from pypgapack import PGA
import numpy as np
import sys

```

```
class MyPGA(PGA) :
    """
    Derive our own class from PGA.
    """
    def tsm(self, p, pop) :
        """
        Oliver's 30 city traveling salesman problem.
        """
        c = self.GetIntegerChromosome(p, pop) # Get pth string as Numpy array
        val = self.distance(c)
        del c
        return val

    def distance(self, c) :
        """
        Compute the total distance for a set of cities.
        """
        # x and y coordinates by city
        x = np.array([54.0,54.0,37.0,41.0, 2.0, 7.0,25.0,22.0,18.0, 4.0,\
                      13.0,18.0,24.0,25.0,44.0,41.0,45.0,58.0,62.0,82.0,\
                      91.0,83.0,71.0,64.0,68.0,83.0,87.0,74.0,71.0,58.0])
        y = np.array([67.0,62.0,84.0,94.0,99.0,64.0,62.0,60.0,54.0,50.0,\
                      40.0,40.0,42.0,38.0,35.0,26.0,21.0,35.0,32.0, 7.0,\
                      38.0,46.0,44.0,60.0,58.0,69.0,76.0,78.0,71.0,69.0])
        n   = self.GetStringLength()
        val = 0.0
        for i in range(0, n-1) :
            val += np.sqrt( (x[c[i]]-x[c[i+1]])**2 + (y[c[i]]-y[c[i+1]])**2 )
        val += np.sqrt( (x[c[0]]-x[c[n-1]])**2 + (y[c[0]]-y[c[n-1]])**2 )
        assert(val > 423.70) # DEBUG.
        return val

    def tbx(self, p1, p2, pop1, c1, c2, pop2) :
        """
        Tie-breaking cross-over.  See Poon and Carter for details.
        """
        # Grab the city id's.
        paren1 = self.GetIntegerChromosome(p1, pop1)
        paren2 = self.GetIntegerChromosome(p2, pop1)
        child1 = self.GetIntegerChromosome(c1, pop2)
        child2 = self.GetIntegerChromosome(c2, pop2)
        assert(np.sum(paren1)==435) # DEBUG
        assert(np.sum(paren2)==435) # DEBUG

        # Copy the parents to temporary vector for manipulation.
        n = self.GetStringLength()
        parent1 = np.zeros(n)
        parent2 = np.zeros(n)
        for i in range(0, n) :
            parent1[i] = paren1[i]
            parent2[i] = paren2[i]

        # Code the parents using "position listing".
```

```
        code1    = np.zeros(n)
        code2    = np.zeros(n)
        for i in range(0, n) :
            code1[parent1[i]] = i + 1
            code2[parent2[i]] = i + 1

        # Randomly choose two cross-over points.
        perm   = np.random.permutation(n)
        point1 = np.min(perm[0:2])
        point2 = np.max(perm[0:2])+1

        # Exchange all alleles between the two points.  (It's unclear to me
        #   whether these points should be inclusive or not; here, they are.)
        temp = np.zeros(point2-point1)
        for i in range(point1, point2) :
            temp[i-point1] = parent1[i]
            parent1[i]     = parent2[i]
            parent2[i]     = temp[i-point1]

        # Generate a cross-over map, a random ordering of the 0,1,...,n-1
        crossovermap = np.random.permutation(n)

        # Multiply each allele of the strung by n and add the map.
        parent1 = parent1*n + crossovermap
        parent2 = parent2*n + crossovermap

        # Replace the lowest allele by 0, the next by 1, up to n-1.  Here,
        #   we sort the parents first, and then for each element, find
        #   where the increasing values are found in the original.  There
        #   is probably a simpler set of functions built in somewhere.
        sort1 = np.sort(parent1)
        sort2 = np.sort(parent2)
        for i in range(0, n) :
            index = np.where(parent1 == sort1[i])
            parent1[index[0][0]] = i
            index = np.where(parent2 == sort2[i])
            parent2[index[0][0]] = i

        # Map the string back to elements.  These are the offspring.
        tempchild1 = np.zeros(n)
        tempchild2 = np.zeros(n)
        for i in range(0, n) :
            tempchild1[parent1[i]] = i
            tempchild2[parent2[i]] = i
        for i in range(0, n) :
            child1[i] = tempchild1[i]
            child2[i] = tempchild2[i]

# Number of cities.
n    = 30

opt = MyPGA(sys.argv, PGA.DATATYPE_INTEGER, n, PGA.MINIMIZE)

```

```python
# One possible benchmark solution
reference = np.array([ 0, 2, 3, 4, 5, 6, 7, 8, 9,10, \
                      11,12,13,14,15,16,17,18,19,20, \
                      21,22,24,23,25,26,27,28,29, 1] )
print "Reference distance is: ", opt.distance(reference)

opt.SetRandomSeed(1)                     # Set seed for verification.
np.random.seed(1)                        # Do the same with Numpy.
opt.SetIntegerInitPermute(0, n-1)        # Start with random permutations.
opt.SetPopSize(400)                      # Large enough to see some success.
opt.SetMaxGAIterValue(100)               # Small number for output.
opt.SetCrossover(opt.tbx)                # Set a cross-over operation.
opt.SetMutation(PGA.MUTATION_PERMUTE)    # Mutate by permutation.
opt.SetUp()                              # Internal allocations, etc.
opt.Run(opt.tsm)                         # Set the objective and run.
opt.Destroy()                            # Clean up PGAPack internals.
```

Running it once yields the following output:

```
***Constructing PGA***
Reference distance is:  423.740563133
Iter #     Field      Value
10         Best       1.012977e+03
Iter #     Field      Value
20         Best       9.924890e+02
Iter #     Field      Value
30         Best       9.924890e+02
Iter #     Field      Value
40         Best       9.924890e+02
Iter #     Field      Value
50         Best       9.419172e+02
Iter #     Field      Value
60         Best       8.709637e+02
Iter #     Field      Value
70         Best       8.709637e+02
Iter #     Field      Value
80         Best       8.524659e+02
Iter #     Field      Value
90         Best       8.524659e+02
Iter #     Field      Value
100        Best       7.805903e+02
The Best Evaluation: 7.805903e+02.
The Best String:
#    0: [      29], [       3], [       0], [      12], [      13], [       9]
#    6: [       4], [       8], [      11], [      10], [       5], [       6]
#   12: [       7], [      17], [      14], [      18], [      16], [      15]
#   18: [      26], [      24], [      22], [      19], [      21], [      25]
#   24: [      27], [      23], [      20], [      28], [       2], [       1]

***Destroying PGA context***
```

2.2.3 Example 6: User-defined Mutation Operator

We redo *Example 2: MAXINT* using a custom mutation operator, largely following the PGAPack example.

```python
"""
pypgapack/examples/example06.py  --  maxint with user mutation.
"""
from pypgapack import PGA
import numpy as np
import sys
class MyPGA(PGA) :
    """
    Derive our own class from PGA.
    """
    def maxint(self, p, pop) :
        """
        The maximum integer sum problem.
        """
        c = self.GetIntegerChromosome(p, pop) # Get pth string as Numpy array
        val = np.sum(c)                       #   and sum it up.
        del c                                 # Delete "view" to internals.
        return float(val)                     # Always return a float.

    def mutate(self, p, pop, pm) :
        """
        Mutate randomly within -n to n
        """
        n     = self.GetStringLength()
        c     = self.GetIntegerChromosome(p, pop)
        count = 0
        for i in range(0, n) :
            if self.RandomFlip(pm) :
                k = self.RandomInterval(1, 2*n)-n
                c[i] = k
                count += 1
        del c
        return count

n   = 10                 # String length.
opt = MyPGA(sys.argv, PGA.DATATYPE_INTEGER, n, PGA.MAXIMIZE)
opt.SetRandomSeed(1)     # Set random seed for verification.
np.random.seed(1)        # Do the same with Numpy.
u_b =  100*np.ones(n)    # Define lower bound.
l_b = -100*np.ones(n)    # Define upper bound.
# Set the bounds.  Note, need to cast as C-combatible integers.
opt.SetIntegerInitRange(l_b.astype('intc'), u_b.astype('intc'))
opt.SetMaxGAIterValue(50)     # 50 generations for short output.
opt.SetMutation(opt.mutate)   # Set a custom mutation.
opt.SetUp()                   # Internal allocations, etc.
opt.Run(opt.maxint)           # Set the objective.
opt.Destroy()                 # Clean up PGAPack internals.
```

Running it yields the following output:

```
***Constructing PGA***
Iter #      Field       Value
10          Best        5.270000e+02
Iter #      Field       Value
20          Best        7.150000e+02
Iter #      Field       Value
30          Best        7.280000e+02
Iter #      Field       Value
40          Best        8.030000e+02
Iter #      Field       Value
50          Best        8.220000e+02
The Best Evaluation: 8.360000e+02.
The Best String:
#     0: [       67], [       65], [       94], [       89], [       85], [       99]
#     6: [       80], [       98], [       87], [       72]

***Destroying PGA context***
```

2.2.4 Example 7: User-defined End of Generation Operator

This example is almost the same as *Example 1: MAXBIT* but it adds an end-of-generation operator. Here, we're using it to flip a random bit to 1. Of course, since we're maximizing the bit sum, this is "climbing the hill" to a better answer. This is a trivial example of such heuristics; in other situations, there are more complex, physically-motivated approaches. Another use of an end-of-generation operator would be for post-generation processing, such as plotting fitnesses, writing to file, etc.

```python
"""
pypgapack/examples/example07.py  --  maxbit with end-of-generation hill climb
"""
from pypgapack import PGA
import sys
class MyPGA(PGA) :
    """
    Derive our own class from PGA.
    """
    def maxbit(self, p, pop) :
        """
        Maximum when all alleles are 1's, and that maximum is n.
        """
        val = 0
        # Size of the problem
        n = self.GetStringLength()
        for i in range(0, n) :
            # Check whether ith allele in string p is 1
            if self.GetBinaryAllele(p, pop, i) :
                val = val + 1
        # Remember that fitness evaluations must return a float
        return float(val)
    def climb(self):
        """
        Randomly set a bit to 1 in each string
```

```
        """
        popsize = self.GetPopSize()
        n = self.GetStringLength()
        for p in range(0, popsize) :
            i = self.RandomInterval(0, n - 1)
            self.SetBinaryAllele(p, PGA.NEWPOP, i, 1)

# (Command line arguments, 1's and 0's, string length, and maximize it)
opt = MyPGA(sys.argv, PGA.DATATYPE_BINARY, 100, PGA.MAXIMIZE)
opt.SetRandomSeed(1)        # Set random seed for verification.
opt.SetMaxGAIterValue(50)   # 50 generations (default 1000) for short output.
opt.SetEndOfGen(opt.climb)  # Set a hill climbing heuristic
opt.SetUp()                 # Internal allocations, etc.
opt.Run(opt.maxbit)         # Set the objective.
opt.Destroy()               # Clean up PGAPack internals
```

Running it yields the following output:

```
***Constructing PGA***
Iter #      Field       Value
10          Best        7.000000e+01
Iter #      Field       Value
20          Best        7.600000e+01
Iter #      Field       Value
30          Best        8.300000e+01
Iter #      Field       Value
40          Best        8.800000e+01
Iter #      Field       Value
50          Best        9.100000e+01
The Best Evaluation: 9.300000e+01.
The Best String:
[ 1111111111111111111011111111111111111011111011011111011111110111 ]
[ 111111111111111111111111101111111111 ]
***Destroying PGA context***
```

Notice that for the same settings, the heuristic improved the best solution a little bit. Of course, flipping just one bit out of 100 shouldn't be expected to work miracles.

2.3 Parallel Examples

The following examples illustrate use of `pypgapack` in a parallel setting using the `mpi4py` package.

2.3.1 Example 8: Parallel MAXBIT

We adapt our favorite *Example 1: MAXBIT* using MPI. We up the string length and population to bring out timing differences.

```
"""
pypgapack/examples/example08.py  --  parallel maxbit
"""
```

```python
from pypgapack import PGA
from mpi4py import MPI
import sys
class MyPGA(PGA) :
    """
    Derive our own class from PGA.
    """
    def maxbit(self, p, pop) :
        """
        Maximum when all alleles are 1's, and that maximum is n.
        """
        val = 0
        # Size of the problem
        n = self.GetStringLength()
        for i in range(0, n) :
            # Check whether ith allele in string p is 1
            if self.GetBinaryAllele(p, pop, i) :
                val = val + 1
        # Remember that fitness evaluations must return a float
        return float(val)

comm = MPI.COMM_WORLD              # Get the communicator.
rank = comm.Get_rank()             # Get my rank.
if rank == 0 :                     # Just to show it works, have node 0
    seed = 1                       #   set seed=1 and n=500 and have all
    n    = 500
else :                             #   other nodes set seed=n=0.  Then,
    seed = 0                       #   broadcast them to all nodes with
    n    = 0
seed = comm.bcast(seed, root=0)  #   node 0 as the root process,
n    = comm.bcast(n,    root=0)  #   and verify by printing.
print " node=", rank, " seed=", seed, " n=", n

if rank == 0 :
    t_start = MPI.Wtime()   # Start the clock.
# (Command line arguments, 1's and 0's, string length, and maximize it)
opt   = MyPGA(sys.argv, PGA.DATATYPE_BINARY, n, PGA.MAXIMIZE)
opt.SetPopSize(1000)
opt.SetRandomSeed(seed)         # Set random seed for verification.
opt.SetMaxGAIterValue(50)       # 50 generations for short output.
opt.SetUp()                     # Internal allocations, etc.
opt.Run(opt.maxbit)             # Set the objective.
opt.Destroy()                   # Clean up PGAPack internals
if rank == 0 :
    t_end = MPI.Wtime()
    print "Elapsed time = ", t_end-t_start, " seconds."
MPI.Finalize() # Should be called automatically, but good practice.
```

Running it using `mpirun -np 1 python example08.py` yields the following output:

```
 node= 0  seed= 1  n= 500
***Constructing PGA***
Iter #     Field      Value
```

```
10         Best       2.930000e+02
Iter #     Field      Value
20         Best       3.030000e+02
Iter #     Field      Value
30         Best       3.080000e+02
Iter #     Field      Value
40         Best       3.240000e+02
Iter #     Field      Value
50         Best       3.280000e+02
The Best Evaluation: 3.290000e+02.
The Best String:
[ 1001010110110110111110111110110010101110101111011000110111000111 ]
[ 1111111011011111111101001011000111011100111111101101100111110001 ]
[ 1010100111101111111111011101011001111111010110011001111111001 ]
[ 0101011111111100011111111101101110110011110111101110110101100000 ]
[ 1011110100011110010110010110011011111001010001001011000101101111 ]
[ 0111101101111100111101011100111111010110111111100101110010110111 ]
[ 1011000101110001111100011111110111110100101110000010010011100010 ]
[ 10111011001011101110011100000111111111111101101111111 ]
***Destroying PGA context***
Elapsed time =  2.06364393234  seconds.
```

Running it using `mpirun -np 2 python example08.py` yields the following output:

```
 node= 0  seed= 1  n= 500
***Constructing PGA***
 node= 1  seed= 1  n= 500
***Constructing PGA***
Iter #     Field      Value
10         Best       2.930000e+02
Iter #     Field      Value
20         Best       3.030000e+02
Iter #     Field      Value
30         Best       3.080000e+02
Iter #     Field      Value
40         Best       3.240000e+02
Iter #     Field      Value
50         Best       3.280000e+02
***Destroying PGA context***
The Best Evaluation: 3.290000e+02.
The Best String:
[ 1001010110110110111110111110110010101110101111011000110111000111 ]
[ 1111111011011111111101001011000111011100111111101101100111110001 ]
[ 1010100111101111111111011101011001111111010110011001111111001 ]
[ 0101011111111100011111111101101110110011110111101110110101100000 ]
[ 1011110100011110010110010110011011111001010001001011000101101111 ]
[ 0111101101111100111101011100111111010110111111100101110010110111 ]
[ 1011000101110001111100011111110111110100101110000010010011100010 ]
[ 10111011001011101110011100000111111111111101101111111 ]
***Destroying PGA context***
Elapsed time =  1.14050102234  seconds.
```

This was using a dual core laptop with probably more browser windows open than needed, but the results look good. Overall, PGAPack focuses parallelism on the object function evaluation. Hence, if your objective function is expensive to evaluate, you can expect relatively good scaling up to the number of strings replaced every generation (the default is 1/10 of the total).

2.3.2 Example 9: Parallel Traveling Salesman

We adapt the Traveling Salesman Problem to parallel and try matching the results Poon and Carter found for the TBX1 cross-over operator. The GA parameters used by Poon and Carter are not entirely clear. They cite results for a population of 21 over 300 iterations with a cross-over to mutation probability ratio of 0.8/0.2. The exact nature of the "swap" mutation is cited from another work I don't have at hand, and the selection appears to be standard elitist, i.e all but the best is replaced. I set a swap operator that always swaps a user-set number of pairs. Using 3 pairs (i.e. 10%) seems to get close to the cited results. Also, because PGAPack doesn't like odd population sizes, I use 22.

```
"""
pypgapack/examples/example09.py  --  traveling salesman in parallel
"""
from pypgapack import PGA
from mpi4py import MPI
import numpy as np
import sys

class MyPGA(PGA) :
    """
    Derive our own class from PGA.
    """
    def tsm(self, p, pop) :
        """
        Oliver's 30 city traveling salesman problem.
        """
        c = self.GetIntegerChromosome(p, pop) # Get pth string as Numpy array
        val = self.distance(c)
        del c
        return val

    def distance(self, c) :
        """
        Compute the total distance for a set of cities.
        """
        # x and y coordinates by city
        x = np.array([54.0,54.0,37.0,41.0, 2.0, 7.0,25.0,22.0,18.0, 4.0,\
                      13.0,18.0,24.0,25.0,44.0,41.0,45.0,58.0,62.0,82.0,\
                      91.0,83.0,71.0,64.0,68.0,83.0,87.0,74.0,71.0,58.0])
        y = np.array([67.0,62.0,84.0,94.0,99.0,64.0,62.0,60.0,54.0,50.0,\
                      40.0,40.0,42.0,38.0,35.0,26.0,21.0,35.0,32.0, 7.0,\
                      38.0,46.0,44.0,60.0,58.0,69.0,76.0,78.0,71.0,69.0])
        n   = self.GetStringLength()
        val = 0.0
        for i in range(0, n-1) :
            val += np.sqrt( (x[c[i]]-x[c[i+1]])**2 + (y[c[i]]-y[c[i+1]])**2 )
        val += np.sqrt( (x[c[0]]-x[c[n-1]])**2 + (y[c[0]]-y[c[n-1]])**2 )
        assert(val > 423.70) # Debug.
```

```
        return val

    def tbx(self, p1, p2, pop1, c1, c2, pop2) :
        """
        Tie-breaking cross-over.  See Poon and Carter for details.
        """
        # Grab the city id's.
        paren1 = self.GetIntegerChromosome(p1, pop1)
        paren2 = self.GetIntegerChromosome(p2, pop1)
        child1 = self.GetIntegerChromosome(c1, pop2)
        child2 = self.GetIntegerChromosome(c2, pop2)
        assert(np.sum(paren1)==435) # DEBUG
        assert(np.sum(paren2)==435) # DEBUG

        # String length.
        n = self.GetStringLength()
        parent1 = np.zeros(n)
        parent2 = np.zeros(n)

        for i in range(0, n) :
            parent1[i] = paren1[i]
            parent2[i] = paren2[i]

        # Code the parents using "position listing".
        code1   = np.zeros(n)
        code2   = np.zeros(n)
        for i in range(0, n) :
            code1[parent1[i]] = i + 1
            code2[parent2[i]] = i + 1

        # Randomly choose two cross-over points.
        perm   = np.random.permutation(n)
        point1 = np.min(perm[0:2])
        point2 = np.max(perm[0:2])+1

        # Exchange all alleles between the two points.
        temp = np.zeros(point2-point1)
        for i in range(point1, point2) :
            temp[i-point1] = parent1[i]
            parent1[i]     = parent2[i]
            parent2[i]     = temp[i-point1]

        # Generate a cross-over map, a random ordering of the 0,1,...,n-1
        crossovermap = np.random.permutation(n)

        # Multiply each allele of the strung by n and add the map.
        parent1 = parent1*n + crossovermap
        parent2 = parent2*n + crossovermap

        # Replace the lowest allele by 0, the next by 1, up to n-1.
        sort1 = np.sort(parent1)
        sort2 = np.sort(parent2)
        for i in range(0, n) :
```

```python
            index = np.where(parent1 == sort1[i])
            parent1[index[0][0]] = i
            index = np.where(parent2 == sort2[i])
            parent2[index[0][0]] = i

        tmpc1 = np.zeros(n)
        tmpc2 = np.zeros(n)
        # Map the string back to elements.  These are the offspring.
        for i in range(0, n) :
            tmpc1[parent1[i]] = i
            tmpc2[parent2[i]] = i
        for i in range(0, n) :
            child1[i] = tmpc1[i]
            child2[i] = tmpc2[i]

    def swap(self, p, pop, pm) :
        """
        Random swap of allele pairs.  Note, nswap must be set!
        """
        n     = self.GetStringLength()
        c     = self.GetIntegerChromosome(p, pop)
        index = np.random.permutation(n)
        for i in range(0, self.nswap) :
            i1 = index[2*i  ]
            i2 = index[2*i+1]
            tmp1     = c[i1]
            tmp2     = c[i2]
            c[i1]    = tmp2
            c[i2]    = tmp1
        del c
        return 0

    def init(self, p, pop) :
        """
        Random initial states.  We do this so that we can enforce the same
        initial guesses for all runs to compare against the Poon and Carter.
        """
        n = self.GetStringLength()
        c = self.GetIntegerChromosome(p, pop)
        np.random.seed(p)
        perm = np.random.permutation(n)
        for i in range(0, n) :
            c[i] = perm[i]
        del c

comm = MPI.COMM_WORLD           # Get the communicator.
rank = comm.Get_rank()          # Get my rank.
t_start = MPI.Wtime()           # Start the clock.
n         = 30                  # Number of cities.
numrun    = 25                  # Number of runs to average.
besteval = np.zeros(numrun)
for i in range(0, numrun) :
    opt = MyPGA(sys.argv, PGA.DATATYPE_INTEGER, n, PGA.MINIMIZE)
```

```
    opt.SetInitString(opt.init) # Set an initialization operator.
    opt.SetCrossoverProb(0.8)   # (Default is 85%)
    opt.SetPopSize(22)          # 22 rather than 21
    opt.SetNumReplaceValue(21)  # Keep the best 22-21 = 1 string = elitist.
    opt.SetMaxGAIterValue(300)  # 300 generations, like the reference.
    opt.SetCrossover(opt.tbx)   # Set a cross-over operation.
    opt.SetMutation(opt.swap)   # Set a mutate operation.
    opt.nswap = 3               # Number of pairs to swap in mutation.
    opt.SetUp()                 # Internal allocations, etc.
    opt.Run(opt.tsm)            # Set the objective and run.
    if rank == 0 :
        best         = opt.GetBestIndex(PGA.OLDPOP)
        besteval[i] = opt.GetEvaluation(best, PGA.OLDPOP)
    opt.Destroy()              # Clean up PGAPack internals.

if rank == 0 :
    print "  MEAN: ", np.mean(besteval) # Print out the mean
    print " SIGMA: ", np.std(besteval)  # and standard deviation.
    print "   MIN: ", np.min(besteval)  # Print out the mean
    print "   MAX: ", np.max(besteval)  # Print out the mean
    t_end = MPI.Wtime()
    print "Elapsed time = ", t_end-t_start, " seconds."

MPI.Finalize() # Should be called automatically, but good practice.
```

Running it using `mpirun -np 1 python example09.py` yields the following output (last several lines):

```
300         Best        9.143837e+02
The Best Evaluation: 9.143837e+02.
The Best String:
#     0: [        7], [       29], [        5], [        2], [       28], [       22]
#     6: [        8], [       10], [       18], [       19], [        3], [       15]
#    12: [       14], [       27], [       16], [        4], [       24], [       20]
#    18: [       17], [       12], [       26], [       25], [        0], [        9]
#    24: [        6], [       13], [       23], [       11], [       21], [        1]

***Destroying PGA context***
  MEAN:  844.962788368
 SIGMA:  97.2494217692
   MIN:  622.625633555
   MAX:  997.810825333
Elapsed time =  67.2148938179  seconds.
```

Running it using `mpirun -np 6 python example09.py` yields the following output (last several lines):

```
Iter #      Field       Value
300         Best        8.004605e+02
The Best Evaluation: 8.004605e+02.
The Best String:
#     0: [        7], [       18], [       17], [       14], [       25], [        5]
#     6: [       19], [       24], [       16], [        2], [        3], [       22]
```

```
#   12: [       21], [        1], [       23], [       26], [        8], [       28]
#   18: [        9], [       12], [       11], [       15], [       20], [       10]
#   24: [       27], [       29], [       13], [        0], [        4], [        6]

***Destroying PGA context***
  MEAN:  832.288188184
 SIGMA:  79.9132513415
   MIN:  597.338150861
   MAX:  942.115139223
Elapsed time =  45.1982860565  seconds.
```

This was using a 6-core machine, but the speedup was barely 25%—why? Well, evaluating the distance of 30 cities is cheap compared to the sorting occuring on the master process for the cross-over operation. As noted earlier, PGAPack's parallelism is definitely meant for expensive evaluations relative to everything else. Here, we see *some* speedup, just not very good. Also compare the mean and standard deviation to the cited 829.00 and 72.69. The parallel results are much closer, which may be a fluke, but it may be worth investigating.

CHAPTER

THREE

METHODS

To be completed.

CHAPTER

FOUR

API REFERENCE

4.1 Introduction

This section provides a reference for all functions defined in the pypgapack module that are *extensions* of the basic PGAPack API. All PGAPack functions are contained in the pypgapack.PGA class. The PGAPack library is typically used as follows:

```
double evaluate(PGAContext *ctx, int p, int pop);
PGAContext *ctx;
ctx = PGACreate(&argc, argv, PGA_DATATYPE_BINARY, 100, PGA_MAXIMIZE);
PGASetUp(ctx);
PGARun(ctx, evaluate);
PGADestroy(ctx);
```

The ctx object is created explicitly by the user and then passed as the first argument to all subsequent function calls, with function names taking the form PGAxxx. For pypgapack, ctx is a *private* member of PGA created during construction, and all PGA members drop the *PGA* prefix and the initial ctx argument. So, for example,

```
ctx = PGACreate(&argc, argv, PGA_DATATYPE_BINARY, 100, PGA_MAXIMIZE);
PGASetUp(ctx);
```

in C/C++ becomes

```
obj = pypgapack.PGA(sys.argv, PGA.DATATYPE_BINARY, 100, PGA.MAXIMIZE)
obj.SetUp()
```

in Python. For all functions included in PGAPack, the user is directed to the pgapack documentation. What follows is a description of the few new methods added for pypgapack that make life in Python a bit easier.

4.2 pypgapack API

The easiest way to see what pypgapack offers is to do the following:

```
>>> import pypgapack as pga
>>> dir(pga)
['PGA', 'PGA_swigregister', '__builtins__', '__doc__', '__file__',
```

```
'__name__', '__package__', '_newclass', '_object', '_pypgapack',
'_swig_getattr', '_swig_property', '_swig_repr', '_swig_setattr',
'_swig_setattr_nondynamic']
```

This command works with any Python module. Our interest is in the `PGA` class. We do the same for this:

```
>>> dir(pga.PGA)
['BinaryBuildDatatype', 'BinaryCopyString', 'BinaryCreateString',
 'BinaryDuplicate', 'BinaryHammingDistance', 'BinaryInitString',
 'BinaryMutation', 'BinaryOneptCrossover', 'BinaryPrint',
 'BinaryPrintString', 'BinaryTwoptCrossover', 'BinaryUniformCrossover',...
```

and find a really long list of class members, most of which are directly from PGAPack. In the following, we document only those not included in PGAPack, as use of the PGAPack functionality is covered above (i.e. drop the `ctx` argument and `PGA` prefix).

class PGA

PGA wrapper class.

`__init__`(*argv*, *datatype*, *n*, *direction*)

Construct the PGA context. This essentially wraps the PGACreate function, so see the PGAPack documentation.

Parameters

- **argv** – system argument
- **datatype** – allele dataype; can be `PGA.DATATYPE_XXX`, where `XXX` is `BINARY`, `INTEGER`, and so on.
- **n** – size of the unknown, i.e. number of alleles of type `datatype`
- **direction** – either `PGA.MAXIMIZE` or `PGA.MINIMIZE`

`GetIntegerChromosome`(*p*, *pop*)

Get direct access to the *p*-th integer chromosome string in population *pop* .

Parameters

- **p** – string index
- **pop** – population index

Returns string as numpy array of integers

`GetRealChromosome`(*p*, *pop*)

Get direct access to the *p*-th double chromosome string in population *pop*.

Parameters

- **p** – string index
- **pop** – population index

Returns string as numpy array of floats

`SetInitString`(*f*)

Set a function for initializing strings. The function `f` provided **must** have the signature `f(p,` `pop)`, but should almost certainly be an inerited class member with the signature `f(self, p, pop)`. See PGAPack documentation for more about user functions.

Parameters f – Python function

See Also:

Example 4: User-defined String Initialization for an example on string initialization.

`SetCrossover`(*f*)

Set a function for the crossover operation. The function `f` provided **must** have the signature `f(a,b,c,d,e,f)`, but should almost certainly be an inerited class member with the signature `f(self,a,b,c,d,e,f)`. See PGAPack documentation for more about user functions.

Parameters f – Python function

See Also:

Example 5: User-defined Crossover Operator for an example on setting the crossover operator.

`SetMutation`(*f*)

Set a function for the mutation operator. The function `f` provided **must** have the signature `f(p, pop, prob)`, but should almost certainly be an inerited class member with the signature `f(self, p, pop, prob)`. See PGAPack documentation for more about user functions.

Parameters f – Python function

See Also:

Example 6: User-defined Mutation Operator for an example on setting the mutation operator.

`SetEndOfGen`(*f*)

Set a function for an operator to be performed at the end of each generation. The function `f` provided **must** have the signature `f(pop)`, but should almost certainly be an inerited class member with the signature `f(self, pop)`. Such an operator can be used to implement hill-climbing heuristics. See PGAPack documentation for more about user functions.

Parameters f – Python function

See Also:

Example 7: User-defined End of Generation Operator for an example on setting the an end of generation operator.

CHAPTER FIVE

LICENSE

`pypgapack` itself is licensed under the MIT license, as follows:

```
Copyright (c) 2011 Jeremy Roberts

Permission is hereby granted, free of charge, to any
person obtaining a copy of this software and associated
documentation files (the "Software"), to deal in the
Software without restriction, including without limitation
the rights to use, copy, modify, merge, publish, distribute,
sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so,
subject to the following conditions:

The above copyright notice and this permission notice shall
be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES
OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
OTHER DEALINGS IN THE SOFTWARE.
```

PGAPack is under its own U-Chicago license that explicitly allows derivatives and redistribution:

```
COPYRIGHT

The following is a notice of limited availability of the code, and disclaimer
which must be included in the prologue of the code and in all source listings
of the code.

(C) COPYRIGHT 2008 University of Chicago

Permission is hereby granted to use, reproduce, prepare derivative works, and
to redistribute to others. This software was authored by:

D. Levine
Mathematics and Computer Science Division
```

```
Argonne National Laboratory Group

with programming assistance of participants in Argonne National
Laboratory’s SERS program.

GOVERNMENT LICENSE

Portions of this material resulted from work developed under a
U.S. Government Contract and are subject to the following license: the
Government is granted for itself and others acting on its behalf a paid-up,
nonexclusive, irrevocable worldwide license in this computer software to
reproduce, prepare derivative works, and perform publicly and display
publicly.

DISCLAIMER

This computer code material was prepared, in part, as an account of work
sponsored by an agency of the United States Government. Neither the United
States, nor the University of Chicago, nor any of their employees, makes any
warranty express or implied, or assumes any legal liability or responsibility
for the accuracy, completeness, or usefulness of any information, apparatus,
product, or process disclosed, or represents that its use would not infringe
privately owned rights.
```

CHAPTER

SIX

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

INDEX