status and future of:
frameworks for optimization &
high-performance computing

Mike McKerns
California Institute of Technology
May 19, 2010

http://dev.danse.us/trac/mystic
overview of major features and goals

• **optimization**
  - optimization with parameter constraints
  - batch & parallel optimization
  - global optimization
  - parameter sensitivity, uncertainty, & correlation
  - domain decomposition
  - configurable convergence and termination conditions

• **high-performance computing**
  - launching and scheduling of computational jobs
  - strategies for distribution of work-list among available resources
  - authentication and tunneling into distributed resources
  - minimal barrier to extending code to high-performance computing
mystic: a simple model-independent inversion framework

About Mystic

The mystic framework provides a collection of optimization algorithms and tools that allows the user to more robustly (and readily) solve optimization problems. All optimization algorithms included in mystic provide workflow at the fitting layer, not just access to the algorithms as function calls. Mystic gives the user fine-grained power to both monitor and steer optimizations as the fit processes are running.

Where possible, mystic optimizers share a common interface, and thus can be easily swapped without the user having to write any new code. Mystic solvers all conform to a solver API, thus also have common method calls to configure and launch an optimization job. For more details, see mystic.abstract_solver. The API also makes it easy to bind a favorite 3rd party solver into the mystic framework.

By providing a robust interface designed to allow the user to easily configure and control solvers, mystic reduces the barrier to implementing a target fitting problem as stable code. Thus the user can focus on building their physical models, and not spend time hacking together an interface to optimization code.

Mystic is in the early development stages, and any user feedback is highly appreciated. Contact Mike McKerns [mmckerns at caltech dot edu] with comments, suggestions, and any bugs you may find. A list of known issues is maintained at http://dev danse us/trac/mystic/query.

Major Features

Mystic provides a stock set of configurable, controllable solvers with::

- a common interface
- the ability to impose solver-independent bounds constraints
- the ability to apply solver-independent monitors
- the ability to configure solver-independent termination conditions
- a control handler yielding: [pause, continue, exit, and user_callback]
- ease in selecting initial conditions: [initial_guess, random]
- ease in selecting mutation strategies (for differential evolution)
This module contains the base class for mystic solvers, and describes the mystic solver interface. The "Solve" method must be overwritten with the derived solver's optimization algorithm. In many cases, a minimal function call interface for a derived solver is provided along with the derived class. See 'mystic.scipy_optimize', and the following for an example.

Usage

A typical call to a mystic solver will roughly follow this example:

```python
>>> # the function to be minimized and the initial values
>>> from mystic.models import rosen
>>> x0 = [0.8, 1.2, 0.7]

>>> # get monitors and termination condition objects
>>> from mystic.tools import Sow
>>> stepmon = Sow()
>>> evalmon = Sow()

>>> from mystic.termination import CandidateRelativeTolerance as CRT

>>> # instantiate and configure the solver
>>> from mystic.scipy_optimize import NelderMeadSimplexSolver
>>> solver = NelderMeadSimplexSolver(len(x0))
>>> solver.SetInitialPoints(x0)
>>> solver.enable_signal_handler()
>>> solver.Solve(rosen, CRT(), EvaluationMonitor=evalmon,
>>>             StepMonitor=stepmon)

>>> # obtain the solution
>>> solution = solver.Solution()
```

An equivalent, yet less flexible, call using the minimal interface is:

```python
>>> # the function to be minimized and the initial values
>>> from mystic.models import rosen
>>> x0 = [0.8, 1.2, 0.7]

>>> # configure the solver and obtain the solution
>>> from mystic.scipy_optimize import fmin
>>> solution = fmin(rosen, x0)
```
Milestone: mystic-0.1a1
Completed 1 year ago

Closed tickets: 22  Active tickets: 0

Initial alpha release.

**Highlights**

**Solvers:**
- Differential Evolution (x2)
- Nelder-Mead Simplex
- Powell's Directional Search Method

**API:**
- solvers share a common interface
- solvers can be called as a unique function or using API
- solvers with built-in optimization control handlers
- configurable solvers can be bound or unbound
- configurable solvers have user-provided or random initial points
- configurable termination conditions
- configurable mutation strategies (for DE solver)

**Tools:**
- configurable 2-variable monitors
- wrap function with counter or bounds
- cost-function generator
- standard set of optimization test models
- set of example scripts for test cases

**Documentation:**
- minimal User's Guide with tutorials
- online Reference Manual

**Issues**

A list of resolved issues is here.
Milestone: **mystic-0.2a1**
Completed 2 months ago

First alpha version for second minor release.

**Highlights**

**Solvers:**
- Pseudo-global Scattershot
- Pseudo-global Batch Grid

**API:**
- parallel mapping of optimization jobs
- distributed parallel mapping of optimization jobs
- parallel multiprocess mapping of optimization jobs
- launching with job schedulers

**Tools:**
- math tools

**Issues**

A list of resolved issues is [here](#).
Milestone: **mystic-0.2a2**

Due in 2 months (08/01/10)

Closed tickets: 0  Active tickets: 14

Second alpha version for second minor release.

**Highlights**

**Solvers:**
- Branch and Bound
- Quasi-Newton BFGS
- Nonlinear Conjugate Gradient

**API:**
- uncertainty quantification
- parameter sensitivity
- parameter correlation
- parameter constraints

**Tools:**
- derivative, gradient, and hessian capture
- performance testing suite
- failure testing suite
- math tools for statistics

**Issues**

A list of resolved issues is [here](#).
Milestone: mystic-0.2a3
Due in 7 months (12/01/10)

Closed tickets: 0  Active tickets: 9

Third alpha version for second minor release.

**Highlights**

Solvers:
- Simulated Annealing
- Noisy BFGS

API:
- improved monitoring and control of optimization jobs
- replace signal handler with event handler

Tools:
- model factories
- discrete measures
- domain decomposition

**Issues**

A list of resolved issues is here.
In-development effort.

Solvers:
- Metropolis
- Least Squares
- Support Vector Machines
- Convex

API:
- programmatic interface to control handler commands
- simultaneous fitting using parameter constraints
- parallel monitoring and control of optimization jobs
- distributed monitoring and control of optimization jobs
- improved control of job schedulers
- launching of solver 'circuit' (parallel, series, etc.)

Tools:
- configurable n-variable monitors
- symbolic math support

Issues
A list of resolved issues is here.
pathos: a framework for heterogeneous computing

About Pathos

Pathos is a framework for heterogenous computing. It primarily provides the communication mechanisms for configuring and launching parallel computations across heterogeneous resources. Pathos provides stagers and launchers for parallel and distributed computing, where each launcher contains the syntactic logic to configure and launch jobs in an execution environment. Some examples of included launchers are: a queue-less MPI-based launcher, a ssh-based launcher, and a multiprocessing launcher. Pathos also provides a map-reduce algorithm for each of the available launchers, thus greatly lowering the barrier for users to extend their code to parallel and distributed resources. Pathos provides the ability to interact with batch schedulers and queuing systems, thus allowing large computations to be easily launched on high-performance computing resources. One of the most powerful features of pathos is "tunnel", which enables a user to automatically wrap any distributed service calls within a ssh-tunnel.

Pathos is divided into four subpackages:

- **dill**: a utility for serialization of python objects
- **pox**: utilities for filesystem exploration and automated builds
- **pyina**: a MPI-based parallel mapper and launcher
- **pathos**: distributed parallel map-reduce and ssh communication

Pathos Subpackage

The pathos subpackage provides a few basic tools to make distributed computing more accessible to the end user. The goal of pathos is to allow the user to extend their own code to distributed computing with minimal refactoring.
Milestone: pathos-0.1a1
Completed 3 weeks ago

Closed tickets: 24  Active tickets: 0

Initial alpha release.

Highlights

Platform Detection and Build:
  • programmatic installation of RPC server to remote user space

Staging and Launching:
  • ssh and scp launchers
  • port selector
  • ssh-tunnel generator
  • mpi launcher
  • parallel python launcher
  • pbs, lsf, and slurm launchers

Communication and Control:
  • serialization of python objects
  • ssh intermediates

Issues

A list of resolved issues is here.
**Milestone: pathos-0.1a2**

*Due in 2 months (08/01/10)*

- Closed tickets: 0  Active tickets: 10

Second alpha version of initial release.

**Highlights**

Platform Detection and Build:
- translation service for remote path and environment variables

Staging and Launching:
- multiprocessing launcher

Communication and Control:
- parallel / distributed mapping strategies
- scheduler monitoring and control
- job kill and recovery

**Issues**

A list of resolved issues is [here](#).
Milestone: **pathos-0.1a3**
Due in 7 months (12/01/10)

Closed tickets: 0  Active tickets: 8

Third alpha version of initial release.

**Highlights**

Platform Detection and Build:
  - remote filesystem exploration and detection
  - repository interactor
  - egg interactor
  - wget interactor
  - zip interactor

Staging and Launching:
  - service factories
  - service factory deployment
  - service deployment

Communication and Control:
  - service management

**Issues**

A list of resolved issues is [here](#).
**Milestone: pathos-dev**

No date set

Closed tickets: 0  Active tickets: 15

In development effort.

Platform Detection and Build:

- programmatic installation of dependencies to remote user space
- programmatic installation of services to remote user space

Staging and Launching:

- service relaunch and recovery
- network detection

Communication and Control:

- node-to-node communication
- publish/subscribe communication
- master job registry

**Issues**

A list of resolved issues is [here](#).
use cases for neutron scattering

- **Rietveld refinement** *(Simon will present at talk end)*
  - traditionally fitting is least-squares, batch refinement with constraints
  - fewer restraints “black magic” by using global optimization
  - large-scale submission of simultaneous refinements

- **Parametric analysis for SANS**
  - fitting analytical shapes to SANS data
  - fitting molecular materials to SANS data

- **Simultaneous model fitting in Reflectometry**
The known crystal structure provides the foundation of the general shape of the scattering object. Only the relative positions in solution of the 3 major subunits (CA, NC, and MA) is in question, allowing an iterative process which probes a severely restricted configuration space for the best fit to the data.

Extracting structure from more complex systems of interest in modern materials and biology is beyond the capability of current generally available tools or the individual investigator to handle. Using coarse graining of known high resolution structures and simulation of scattering from arbitrarily drawn shapes, provides one mechanism for extracting information from scattering data from such materials.
Interaction of Alzheimer’s peptide amyloid β with lipid bilayer membranes

M. Loesche (Carnegie Mellon University), M. Doucet, P. A. Kienzle (University of Maryland)

Amyloid plaques associated with dead or damaged neurons are a characteristic property of patients suffering from Alzheimer’s disease. A major component of these plaques is the amyloid β peptide (1-42) which plays a key role in the progress of the disease. Two leading hypotheses exist: either amyloid β forms ion channels in the membrane or amyloid β leads to membrane thinning and disruption. Neutron reflectivity, which is able to test both hypotheses by measuring membrane thickness and completeness in presence of amyloid β, favors the second hypothesis. The layer densities and thicknesses were obtained by simultaneously fitting of all datasets using DANSE reflectometry software. The DANSE project aims to exploit the natural parallelism of the genetic algorithm search strategy and improve the user interface so that results can be obtained more quickly and easily.

Figure 1: Neutron reflectivity curves measured on a tethered lipid membrane with and without amyloid β. The results clearly show a fully reversible membrane thinning in the presence of amyloid β.
vnf: optimization of neutron experiments
mystic: optimizers & the solver API

- Steepest-descent optimizers
  - Nelder-Mead Simplex
  - Powell’s Directional Search
  - Quasi-Newton BFGS
  - Nonlinear Conjugate Gradient

- Global optimizers
  - Differential Evolution
  - Branch and Bound
  - Simulated Annealing

- the Abstract Solver Interface
  
  ```python
  Solve(costfunction, termination, sigint_callback=None,
       EvaluationMonitor=Null, StepMonitor=Null, ExtraArgs=(), **kwds)
  ```
Requirements for building a cost function

- Model should be structured as a function call
  \[ \text{Fx} = \text{model}(\text{params}) \]

- User must provide a difference metric
  \[ \text{cost} = (\text{Fx} - \text{target})^2 \]

- Cost function must be callable from Python

- Input is a list of model parameters; output is a single cost value

Example of dynamic generation of a cost function:

```python
# generate cost function for model & data
def costFactory(model, target):
    
    # evaluate cost of given input parameters
def cost(params):
        Fx = model(params)
        return (Fx - target)**2

    # return cost function
return cost
```
scenarios for cost function analysis

- **Definitions**
  - $F$ is model; $F(x)$ is model evaluated at some fitted parameter set
  - $G$ is target data; $G(p)$ is series data at some set of system parameters
  - $F(p)$ is model evaluated at series data points

- **Optimization**
  \[ cost := \min(F(x) - G)^2 \]

- **Parameter Sensitivity**
  \[ cost := -\min(F(x) - F(y))^2 \text{ with } x_j = y_j \text{ for } j \neq i \]

- **Model Validation**
  \[ cost := -\min(F(p) - G(p))^2 \]
measures of parameter sensitivity

- \( D_i[F] := \sup \{ |F(x) - F(y)| : x_j = y_j \text{ for } i = j \} \)
  - the diameter \( D \) of a function \( F \) measures the model variability over the range of given input parameters
  - diameter evaluations require the solution of a global optimization problem over the range of the inputs (define \( \text{cost} := D_i[F]^2 \))

- \( D[F] := (D_1[F]^2 + \ldots + D_N[F]^2)^{1/2} \)
  - each independent variable has a sub-diameter \( D_i \) which can all be collected to provide a single measure of parameter impact on the model

- \( D_i[F] / D[F] \)
  - provides normalized measure of impact of a single parameter

http://www.cacr.caltech.edu/~mmckerns/uq_mystic
partitioning to find ‘interesting’ regions of parameter space
- calculate PoF upper bound for each region of parameter space
- identify regions where PoF = 1 or 0; remove as ‘uninteresting’
- select region with highest contribution to the PoF upper bound
- divide parameter space along the axis with largest diameter

discovery of regions of critical behavior
user’s view of optimization

```python
# select monitors to log the current coeffs & results… sow(coeffs,result)
from mystic.tools import Sow
stepmon = Sow()

# select termination to manage solver convergence… T or F = condition(solver instance)
from mystic.termination import CandidateRelativeTolerance as CRT
termcond = CRT()

# import a model that takes a list of coeffs… result = function(coeffs)
from mystic.models import rosen
x0 = [0.8, 1.2, 0.7]

# select optimizer… prepare to accept the model & termination conditions
from mystic.differential_evolution import DifferentialEvolutionSolver2 as DESolver
solver = DESolver(len(x0))

# configure and launch solver
solver.SetInitialPoints(x0)
solver.Solve(rosen, termcond, StepMonitor=stepmon)
solution = solver.Solution()
```
new methods for service-based computing

# RPC-based parallel mapping

def SelectServers(self, servers, ncpus=None):
    """Takes tuple of ('hostname:port'), listing each available compute server""
    self._servers = servers; self._ncpus = ncpus
    return

# MPI-based parallel mapping

def SetMapper(self, mapper):
    """Set mapping strategy. Takes a mapping function""
    self._mapper = mapper
    return

def SetLauncher(self, launcher, nnodes=None):
    """Set launcher and nodes. Takes a launcher function and (optionally) # of nodes""
    self._launcher = launcher; self._nnodes = nnodes
    return
example: the differential evolution solver

```python
# the generalized solver algorithm
for generation in range(self._maxiter):

    StepMonitor(self.bestSolution[:], self.bestEnergy)

    ...<generate a list of trial solutions trialPop> ...

    trialEnergy = map(costfunction, trialPop)

    ... <check if energy of trial solutions are lower than the current best energy> ...

    if self._EARLYEXIT or termination(self):
        break

    ...
    return

# here “map” is just python’s map function
# however, it is a natural interface to lots of underlying complexity in job management
```
parallel differential evolution solver

# the generalized solver algorithm
for generation in range(self._maxiter):

    # StepMonitor(self.bestSolution[:,], self.bestEnergy)

    …<generate a list of trial solutions trialPop> …

    trialEnergy = map(costfunction, trialPop, processes=self._ncpus, servers=self._servers)

    … <check if energy of trial solutions are lower than the current best energy> …

if self._EARLYEXIT or termination(self):
    break

…
return

# here “map” is the RPC-based parallel_map (or alternately, the MPI-based ez_map)
# otherwise the solver code is identical
mystic as a parallel fitting service

- Map hides complexity of underlying service connections
  - cost function cast as a distributed service (i.e. on another machine)
  - implies management of service communications (with job manager)
distributed & parallel computing

- **Parallel computing**
  - Each model may require its own parallel compute cluster.

- **Distributed computing**
  - Multiple copies of models working together for global optimization and in parameter sensitivity studies.
  - Queue configuration and launch commands must be passed between compute resources.
key elements of the software architecture

• the Job Manager
  - stages and launches new jobs
  - broadcasts execution control directives
  - maintains registry of submitted jobs

• the Iterator
  - adjusts cost function parameters
  - reacts to control directives

• the Mapping Strategy
  - provides algorithm to distribute workload among available resources

• the Launcher
  - knows how to submit jobs on the current execution environment
Mapper abstracts the underlying distributed services
- distributes workload among available resources
- submits jobs on the selected execution environment
parallel launchers & workflow controls

- transparent selection of the job launcher
  - standard python (i.e. serial)
  - parallel python
    - multiprocessor, parallel, or distributed
    - mixed (of the above three)
  - MPI python
    - queue-less parallel or queued parallel

- dynamic control of running fit jobs
  - interrupt / continue
  - exit: kill current fit job
  - sol: get current best solution
  - call: execute user-provided callback function
pathos: tools for heterogeneous computing

- **pyina: parallel launchers & mappers**
  - required for each environment and execution mechanism
  - provide different strategies for processing a work-list
  - utilized by the mystic solvers

- **dill: a full python state pickler**
  - serialization of nearly all python types
  - provided as a standalone package

- **pox: environment / filesystem exploration and build**
  - pure-python versions of unix filesystem query and build commands
  - leverage deploy and build tools: *make, svn, python eggs, …*
  - provided as a standalone package
# ssh Launcher
launcher = LauncherSSH('LauncherSSH')
launcher.stage(command=command1, rhost=rhost, fgbg='background')
launcher.launch()
print launcher.response()

# scp Launcher
copier = LauncherSCP('LauncherSCP')
print 'executing {scp %s:%s %s:%s}' % (cpu1,source1,cpu2,dest1)
copier.stage(source=cpu1+':'+source1, destination=cpu2+':'+dest1)
copier.launch()

# ssh Tunnel builder
t = Tunnel('Tunnel')
lport = t.connect(rhost, rport)
# …<do stuff>…
t.disconnect()

# port selection
# lport = portnumber(min=1024,max=65535)()
rport = pickport(rhost)
deployment of a distributed service factory

Base service is a foothold on the distributed resource. It should not be lightweight. It will be responsible for launching jobs, maintaining a connection to its children, and keeping a registry of jobs.

Service spawned as a daemon
deployment of a distributed service

Base Service is a local manager for spawned services. Management requests come through the ssh-tunnel.

<table>
<thead>
<tr>
<th>name</th>
<th>pid</th>
</tr>
</thead>
<tbody>
<tr>
<td>name#1</td>
<td>pid#1</td>
</tr>
</tbody>
</table>

Spawning Service handles compute requests through ssh-tunnel. Is managed through direct connection to Base Service.
accomplished goals for SNS deployment

• initial “test” deployment on SNS resources
  - pathos: distributed and high-performance computing framework
  - mystic: optimization framework
  - extend DANSE applications to high-performance computing
    - VNF: virtual neutron facility
    - SrRietveld: parallel batch Rietveld refinements

• no new user requirements for SNS/NSSD users
  - XCAMS account; access to computing resources
  - https://neutronsrsr.us/accounts

• no new cyber-security policy requirements
  - authentication with ssh-keypair matching
  - connections between user’s own accounts (& approved group accounts?)
key tests performed on SNS resources

• Test 1: Optimization
  - Install mystic on SNS resources
  - Confirm with a function minimization

• Test 2: Distributed Computing
  - Install pathos on SNS resources
  - Connect with ssh-tunneled RPC server
  - Confirm with workload-balanced calculation of prime numbers

• Test 3: High-Performance Computing
  - Install parallel mystic on SNS resources
  - Connect with MPI & slurm submission
  - Confirm with parallel-mapped calculation of prime numbers
what’s next?

• Install “high-performance” releases at SNS
  - complete the release documentation… and deploy with “easy_install”
  - provides both standard and parallel mystic solvers
  - provides parallel and distributed batch submission

• Flagship applications utilize mystic and pathos
  - VNF (e.g. for sample kernel job submission)
  - global optimization (e.g. for Rietveld refinement)
  - parallel batch submission

• Merge development branch features to Y5 releases
  - also merge features from PARK, Refl*, and SrFit

• Usage testing at the SNS
y5 and after the DANSE…

• grow the already broad community usage base
  - major public releases in y5 (mystic 4th-6th; pathos 2nd-4th)
  - publications (already at least 4 using mystic)
  - improve on the 100’s of downloads to unique IPs

• plan for sustainability
  - mystic & pathos used by other large-scale computing projects at Caltech
  - key component in two proposals for new funding

• strong commitment to NSSD user base
  - support NSSD resource requirements
  - work with users on developing new science cases / software
End Presentation