The optimal uncertainty algorithm in the *mystic* framework

M. McKerns*, H. Owhadi†, C. Scovel‡, T. J. Sullivan§, & M. Ortiz¶

August 21, 2010

Abstract

We have recently proposed a rigorous framework for Uncertainty Quantification (UQ) in which UQ objectives and assumption/information set are brought into the forefront, providing a framework for the communication and comparison of UQ results. In particular, this framework does not implicitly impose inappropriate assumptions nor does it repudiate relevant information.

This framework, which we call *Optimal Uncertainty Quantification* (OUQ), is based on the observation that given a set of assumptions and information, there exist bounds on uncertainties obtained as values of optimization problems and that these bounds are optimal. It provides a uniform environment for the optimal solution of the problems of validation, certification, experimental design, reduced order modeling, prediction, extrapolation, all under aleatoric and epistemic uncertainties.

OUQ optimization problems are extremely large, and even though under general conditions they have finite-dimensional reductions, they must often be solved numerically. This general algorithmic framework for OUQ has been implemented in the *mystic* optimization framework. We describe this implementation, and demonstrate its use in the context of the Caltech surrogate model for hypervelocity impact.

*Materials Science, California Institute of Technology, Pasadena, CA 91125, USA. Email: mmckerns@caltech.edu

†Applied & Computational Mathematics, California Institute of Technology, Pasadena, CA 91125, USA. Email: owhadi@caltech.edu

‡Modeling, Algorithms, & Informatics, Los Alamos National Laboratory, Los Alamos, NM 87545, USA. Email: jcs@lanl.gov

§Graduate Aerospace Laboratories, California Institute of Technology, Pasadena, CA 91125, USA. Email: tjs@caltech.edu

¶Graduate Aerospace Laboratories, California Institute of Technology, Pasadena, CA 91125, USA. Email: ortiz@aero.caltech.edu

1 The OUQ algorithm in the *mystic* framework

The *mystic* optimization framework [1] provides a collection of optimization algorithms and tools that lowers the barrier to solving complex optimization problems. *Mystic* provides a selection of optimizers, both global and local, including several gradient solvers. The optimizers included in *mystic* all use a common interface, so they can be easily interchanged without the user having to write new code.

Constraints in *mystic* are classified as either "bounds constraints" (linear inequality constraints that involve precisely one input variable) or "non-bounds constraints" (constraints between two or more parameters). Every *mystic* optimizer provides the ability to apply bounds constraints generically and directly to the cost function, so that the difference in the speed of bounds-constrained optimization and unconstrained optimization is minimal. *Mystic* also enables the user to impose an arbitrary parameter constraint function on the input of the cost function — allowing non-bounds constraints to be generically applied in any optimization problem.

In addition to standard penalty methods that couple the solution of the constraints with the solution of the optimization problem, *mystic* provides methods for imposing constraints on a discrete set. Both the statistical methods and the constraints solvers in *mystic* are thus built to decouple the constraints from the optimization problem. This implementation greatly reduces the complexity of the optimization problem, and the number of function evaluations does not blow up with the complexity of constraints. Since evaluation of the model is commonly the most expensive part of the optimization, *mystic* optimizers should also converge much more quickly than other algorithms that apply constraints by invalidating generated results (*i.e.* filtering) at each iteration. In this way, *mystic* can efficiently solve for rare events because the set of input variables produced by the optimizer at each iteration will also be an admissible point in problem space — this feature is critical in solving OUQ problems, as tens of thousands of function evaluations may be required to produce a solution. Our implementation of the OUQ algorithm in *mystic* utilizes a nested optimization (*i.e.* the OUQ inner loop) to solve an arbitrary set of parameter constraints at each evaluation of the cost function. Hence, the OUQ algorithm depends on reliably obtaining accurate solutions to the constraints at each iteration. Fortunately, *mystic* provides solver-independent termination conditions — a capability that greatly increases the flexibility for numerically solving problems with non-standard convergence profiles.

The implemention of OUQ within the *mystic* framework is presented below.

OUQ as an optimization problem. OUQ problems can be thought of optimization problems where the goal is to find the global maximum of a probability function $\mu[H = 0]$, where $H = 0$ is a failure criterion for the model response function H. Additional conditions in an OUQ problem are provided as constraints on the information set. Typically, a condition such as a mean constraint on H, $m_1 \leq \mathbb{E}_\mu[H] \leq m_2$, will be imposed on the maximization. After casting the OUQ problem in terms of optimization and constraints, we can plug these terms into the infrastructure provided by *mystic*.

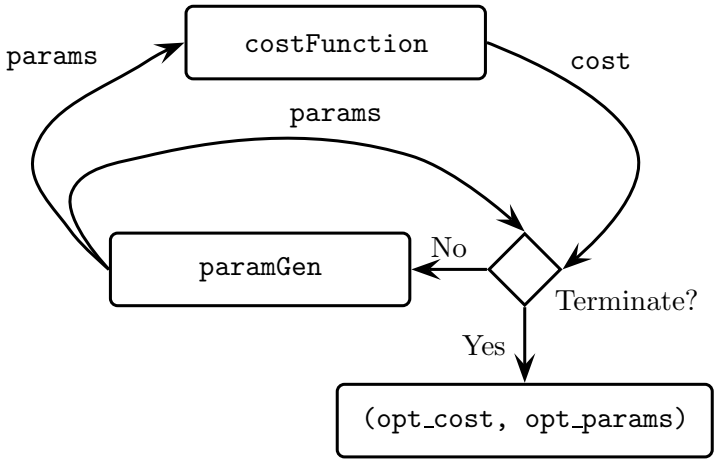


Figure 1.1: Conceptual diagram of an optimization loop. Parameters `params` are generated by the optimizer's parameter generator `paramGen` each iteration, and are piped into the cost function `costFunction`. The cost function is evaluated at the model parameters and returns the `cost`. The convergence criteria are checked after every iteration, and if the termination conditions are met, the optimizer stops. `opt_cost` is the resulting calculated maximum, and `opt_params` are the calculated maximizers.

In the most general sense, optimization is a feedback loop between a measure of "goodness of fit" (or `cost`) for a chosen model and the optimization algorithm (see Figure 1.1). The `costFunction` is a measure of quality that evaluates the `cost` for a set of model parameters `params`. The natural metric for $\mathbb{P}[H = 0]$ is the optimization criteria, hence we select the evaluation of $\mu[H = 0]$ as the `costFunction`. The optimizer will generate trial parameters every iteration in the feedback loop until the termination conditions are met. Termination is typically decided by convergence criteria on the last n values of `cost` and `params`, however it can also be determined by conditions on the current values of `cost` and `params`, the number of `costFunction` evaluations, the number of iterations of the feedback loop, or any combinations thereof. Upon satisfying the termination conditions, the optimizer produces the calculated maximal $\mu[H = 0]$ as `opt_cost` and the calculated maximizers as `opt_params`.

Measures as data objects. It is noted that a solution to an OUQ problem can be expressed in terms of product measures [2]. Thus, in a computational implementation of OUQ, the natural means for passing information between different elements of the algorithm code should also be based on product measures. A hierarchy of parameterized measure data objects is the natural commodity for the evaluation of $\mu[H = 0]$. The optimizer's parameter generator `paramGen` will produce new parameters each iteration, and hence produce new product measures to be evaluated within the `costFuntion`. For instance, a response function H that requires input of dimension $n = 3$, here defined by

$H(x, y, z)$, will require a product measure of dimension $n = 3$ for support. In problems with bounds constraints on the mean of the response function, such as $[m_1, m_2] = [0, 1]$, where the parameters x, y, z may also be bounded by range, we then can use products of convex combinations of Dirac masses as the basis for support. The corresponding OUQ code parameterizes the Dirac masses by their weights and positions. These weights and positions become the inputs for the optimization problem, and the solved weights and positions are thus the maximizers for the optimization problem.

The following general formulation is applicable to product measures of any dimension, and also for measures that utilize a basis other than Dirac masses. However, for simplicity the following is described in the context of an optimization over Dirac masses of dimension $n = 3$.

We use the Dirac measure class, available from *mystic*, to build a dimension $n = 3$ product measure data object hierarchy that supports the transfer of information through the optimization loop (see Figure 1.1), where:

- a 3D product measure is composed of 3 one-dimensional discrete measures — this construction is the computational representation of the tensorization of measures to form a product measure;
- these one-dimensional discrete measures are data objects composed of N "support points" — these objects represent (not-necessarily-convex) linear combinations of weighted Dirac measures;
- support points are data objects that have mass and position — these objects represent weighted Dirac measures on the one-dimensional input parameter spaces.

Discrete measures are provided with the methods:

- "coords": set or return the positions of each support point;
- "weights": set or return the mass of each support point;
- "npts": return the number of support points;
- "mass": calculate the sum of all weights;
- "normalize": set the sum of all weights to 1.0;
- "range": set or calculate "max − min" for the positions;
- "mean": set or calculate the centre of mass of the measure, *i.e.* the Euclidean dot product of the "coords" array with the "weights" array.

As mentioned above, discrete measures are composed of finitely many support points, with each support point having weight and position. The first three of the above methods are required to describe the composition of the measure itself, while the last four methods provide the measure with additional mathematical properties. It is worth noting that the implementation of "normalize", "range", and "mean" are such that setting one of these three properties does not alter the value of the other two. For example, imposing a mean is done by translating the positions of all basis elements in a discrete measure equally, thereby preserving the measure's range and mass. This implementation is an essential part of helping *mystic* decouple constraints from the optimization problem.

Additionally, constraints on the range of the input parameters are applied directly to the cost function using the `function_wrapper` method provided by *mystic*. These parameter range constraints ensure that the weights are in the interval $[0, 1]$, while an additional constraints function enforces the normalization (total measure = 1) condition by calling "normalize".

Product measures are provided with the methods:

- "coords": set or return the positions of each support point
- "weights": return the mass of each support point
- "npts": return the number of support points
- "expect": set or calculate $\mathbb{E}_\mu[H]$ for response function H

While product measures require the same three methods to describe the composition of the measure as discrete measures do, product measures are provided with one new property, the method "expect", which sets or calculates the expectation value. A OUQ algorithm implementation requiring a mean constraint on H thus defines a constraint function that ensures that $m_1 \leq E_\mu[H] \leq m_2$, and additionally that the sum of the weights on each discrete measure is equal to 1.0. Additional constraints on the information set are added to the optimization problem by adding conditions within the constraints function implemented using the above methods.

Also, several helper utilities are available to aid in conversion between the different data object representations:

- "unpack": maps a n dimensional product measure to its n factors, which are 1D discrete measures;
- "pack": map n 1D discrete measures to one dimension n product measure, *i.e.* forms their product measure;
- "flatten": map n x 1D discrete measures to an input parameter list;
- "unflatten": map an input parameter list to n x 1D discrete measures.

The above implementation allows constraints to be applied in terms of their natural data representation. For example, to apply mean constraints to a product measure, we would first break the product measure into its component discrete measures with "unpack", use "mean" to impose a mean on the relevant 1D discrete measures, and then use "pack" to build the new product measure. Since constraints are applied on measure objects, but a `costFunction` requires a parameter list `param` as input, "flatten" must be used to convert measure objects to a list of weights and positions. After the optimizer provides the new `param` values, "unflatten" then converts the modified list of weights and positions into new measure objects.

It is worthwhile to note that the OUQ algorithm presented in the remainder of this subsection is independent of the specific implementation of the measure data objects. As shown in Figure 1.1, a list of `params` and a `cost` is sufficient to exchange information between portions of the optimization loop. A more complex object hierarchy is required

only within the `costFunction`, and in the application of the non-bounds constraints (which utilize measure object methods). By providing the data object hierarchy with "flatten" and "unflatten" methods, the measure data objects can be decoupling from the rest of the algorithm. All bounds constraints are directly applied to the cost function, and act on a single variable (*i.e.* a member of the `params` list), while all non-bounds constraints are applied within the constraint function, which takes `params` as both input and output (as shown in the text below). Thus, the product measure hierarchy that is chosen to best provide support for the model, whether it be a convex combination of Dirac masses, or measures composed of another basis such as Gaussians, has no effect on the implementation of the remainder of the OUQ algorithm code.

More precisely, since measures are handled as data objects, this framework can naturally be extended to Gaussians by adding covariance matrices as data object variables and by estimating integral moments equations (with a Monte Carlo method for instance) instead of using the equations given below. Indeed, *mystic* also provides Gaussian measure objects which can be selected for OUQ optimizations.

In the basic situation in which Dirac measures are the basis of the 1D measures (as opposed to, say, Gaussians), the optimizer can discover the weights and positions of the Diracs that maximize $\mu[H = 0]$. Note that, when μ is a product of convex combinations of Dirac measures, *i.e.*

$$\mu = \left(\sum_{i=1}^{N_x} w_{x_i} \delta_{x_i} \right) \otimes \left(\sum_{j=1}^{N_y} w_{y_j} \delta_{y_j} \right) \otimes \left(\sum_{k=1}^{N_z} w_{z_k} \delta_{z_k} \right), \tag{1.1}$$

the μ-probability of failure $\mu[H = 0]$ and the mean value of the response function $\mathbb{E}_\mu[H]$ can be easily calculated as follows:

$$\mu[H = 0] = \sum_{i,j,k} w_{x_i} w_{y_j} w_{z_k} \mathbb{1}[H(x_i, y_j, z_k) = 0], \tag{1.2}$$

$$\mathbb{E}_\mu[H] = \sum_{i,j,k} w_{x_i} w_{y_j} w_{z_k} H(x_i, y_j, z_k). \tag{1.3}$$

2 A motivating physical example and associated results

In this section, we discuss the results of numerical implementation of OUQ algorithms for a simple surrogate model for hypervelocity impact. In particular, we compute the upper ($\mathcal{U}(\mathcal{A})$) bound on the probability of failure for various sets of assumptions $\mathcal{A}$, and also consider a simple example of the experimental selection problem.

The physical system of interest is one in which a 400C steel ball of diameter $D_\mathrm{p} = 1.778\,\mathrm{mm}$ impacts a 440C steel plate of thickness h (expressed in mm) at speed v (expressed in $\mathrm{km} \cdot \mathrm{s}^{-1}$) at obliquity θ from the plate normal. The physical experiments are performed at the California Institute of Technology SPHIR (Small Particle Hypervelocity Impact Range) facility. A simple surrogate model was developed to calculate

the perforation area (in mm^2) caused by this impact scenario. The surrogate response function is as follows:

$$H(h, \theta, v) = K \left(\frac{h}{D_\mathrm{p}} \right)^p (\cos\theta)^u \left(\tanh\left(\frac{v}{v_\mathrm{bl}} - 1 \right) \right)_+^m, \tag{2.1}$$

where the *ballistic limit velocity* (the speed below which no perforation area is caused) is given by

$$v_\mathrm{bl} := H_0 \left(\frac{h/\mathrm{mm}}{(\cos\theta)^n} \right)^s. \tag{2.2}$$

The seven quantities H_0, s, n, K, p, u and m are fitting parameters that have been chosen to minimize the least-squares error between the surrogate and a set of 56 experimental data points; they take the values

$$\begin{aligned} &H_0 = 0.5794\,\mathrm{km}\cdot\mathrm{s}^{-1}, && s = 1.4004, && n = 0.4482, && K = 10.3936\,\mathrm{mm}^2, \\ &p = 0.4757, && u = 1.0275, && m = 0.4682. \end{aligned}$$

For the remainder of this section, the surrogate response function, H, will be taken as given and fixed, and its behaviour will be investigated on the input parameter range

$$(h, \theta, v) \in \mathcal{X} := \mathcal{X}_1 \times \mathcal{X}_2 \times \mathcal{X}_3, \tag{2.3a}$$

$$h \in \mathcal{X}_1 := [1.524, 2.667]\,\mathrm{mm} = [65, 105]\,\mathrm{mils}, \tag{2.3b}$$

$$\theta \in \mathcal{X}_2 := [0, \tfrac{\pi}{6}], \tag{2.3c}$$

$$v \in \mathcal{X}_3 := [2.1, 2.8]\,\mathrm{km}\cdot\mathrm{s}^{-1}. \tag{2.3d}$$

We will measure lengths in both mm and mils (recall that $1\,\mathrm{mm} = 0.0254\,\mathrm{mils}$). Since H has been fixed, the optimizations that follow are all optimizations with respect to the unknown random distribution $\mathbb{P} \in \mathcal{M}(\mathcal{X})$ of the inputs of H. We adopt the "gunner's perspective" that the failure event is non-perforation, *i.e.* the event $[H = 0]$.

Computation of optimal bounds on the probability of failure. We assume that the impact velocity, impact obliquity and plate thickness are independent random variables, and that the mean perforation area must lie in a prescribed range $[m_1, m_2] := [5.5, 7.5]\,\mathrm{mm}^2$. Therefore, the corresponding admissible set for the OUQ problem is

$$\mathcal{A} := \left\{ (H, \mu) \;\middle|\; \begin{array}{l} H \text{ given by (2.1)}, \\ \mu = \mu_1 \otimes \mu_2 \otimes \mu_3, \\ m_1 \leq \mathbb{E}_\mu[H] \leq m_2 \end{array} \right\}. \tag{2.4}$$

With access to H, bounds on the probability of non-perforation can be calculated. In order to find the optimal upper bound, $\mathcal{U}(\mathcal{A})$, it is sufficient to search among those measures μ whose marginal distributions on each of the three input parameter ranges

have support on at most two points [2]. That is, $\mathcal{U}(\mathcal{A}) = \mathcal{U}(\mathcal{A}_\Delta)$, where the reduced feasible set $\mathcal{A}_\Delta$ is given by

$$\mathcal{A}_\Delta := \left\{ (H, \mu) \;\middle|\; \begin{array}{c} H \text{ given by (2.1)}, \\ \mu = \mu_1 \otimes \mu_2 \otimes \mu_3, \\ \mu_i \in \Delta_1(\mathcal{X}_i) \text{ for } i = 1, 2, 3, \\ m_1 \leq \mathbb{E}_\mu[H] \leq m_2 \end{array} \right\}. \tag{2.5}$$

A numerical optimization over this finite-dimensional reduced feasible set $\mathcal{A}_\Delta$ using a genetic algorithm global optimization routine in the *mystic* framework yields the following optimal upper bound on the probability of non-perforation:

$$\mathbb{P}[H = 0] \leq \mathcal{U}(\mathcal{A}) = \mathcal{U}(\mathcal{A}_\Delta) \overset{\text{num}}{=} 37.9\%.$$

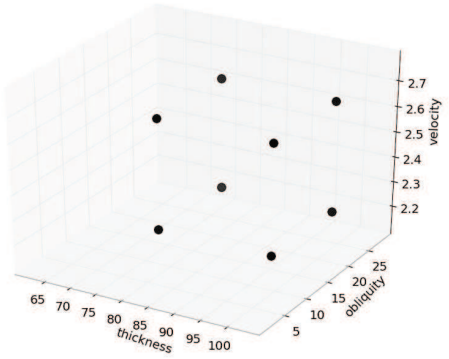
Observe that $\mathbb{P}[H = 0] \leq \mathcal{U}(\mathcal{A}) = \mathcal{U}(\mathcal{A}_\Delta)$ is a theorem whereas the $\mathcal{U}(\mathcal{A}_\Delta) \overset{\text{num}}{=} 37.9\%$ is the output of an algorithm (in this case, a Differential Evolution Algorithm implemented in the *mystic* framework, see Section 3). In particular, its validity is correlated with the efficiency of the specific algorithm. We have introduced the symbol $\overset{\text{num}}{=}$ to emphasize the distinction between mathematical equalities and numerical outputs.

Although we don't have a theorem associated with the convergence of the numerical optimization algorithm, we have a robust control over its efficiency because it is applied to the finite dimensional problem $\mathcal{U}(\mathcal{A}_\Delta)$ instead of the infinite optimization problem associated with $\mathcal{U}(\mathcal{A})$.

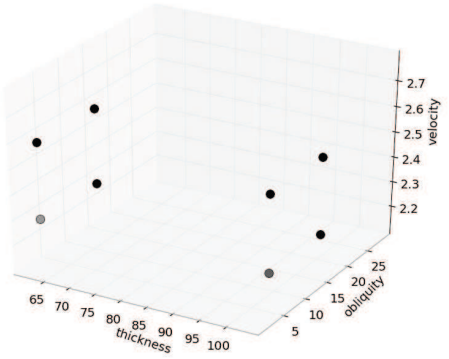
For $\#\text{supp}(\mu_i) \leq 2$, $i = 1, 2, 3$ (where $\#\text{supp}(\mu_i)$ is the number of points forming the support of μ_i), Figure 2.1 shows that numerical simulations collapse to two-point support. Indeed, even when a wider search is performed (*i.e.* over measures $\mu \in \bigotimes_{i=1}^{3} \Delta_k(\mathcal{X}_i)$ for $k > 1$), it is observed that the calculated maximizers for these problems maintain two-point support: the velocity and obliquity marginals each collapse to a single Dirac mass, and the plate thickness marginal collapses to have support on the two extremes of its range. As expected, optimization over a larger search space is more computationally intensive and takes longer to perform. We also refer to Figure 2.2 for plots of the locations and weights of the Dirac masses forming each marginal μ_i as functions of the number of iterations. Note that the lines for *thickness* and *thickness weight* are of the same color if they correspond to the same support point for the measure.

3 Implementation of the physical example in *mystic*

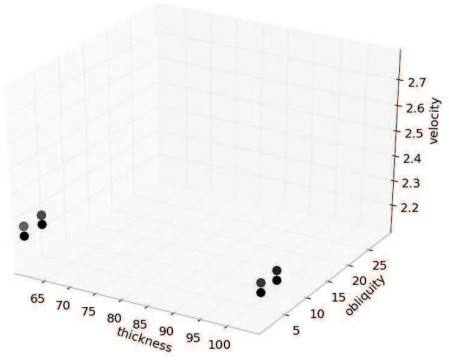
The OUQ outer loop. As posed above, an OUQ problem at the high-level is a global optimization of a cost function that satisfies a set of constraints. The optimization of the example presented in Section 2 is performed in *mystic* using the differential evolution algorithm of Price & Storn [3, 4]. Differential evolution is a form of genetic algorithm that generates `npop` trial solutions each iteration (here `npop` is called the "trial population size"), and uses a "mutation strategy" to modify trial solutions in attempt to produce a better solution. Beyond selecting the mutation strategy, two parameters



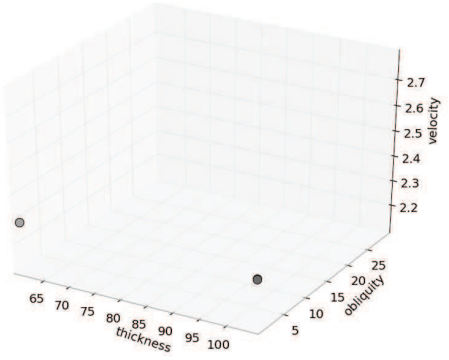
(a) support points at iteration 0



(b) support points at iteration 150

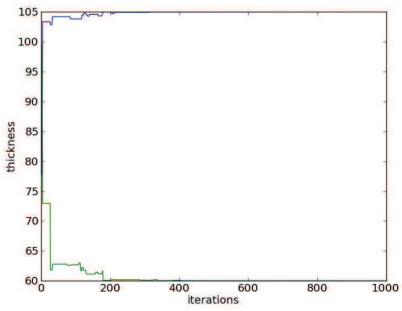


(c) support points at iteration 200

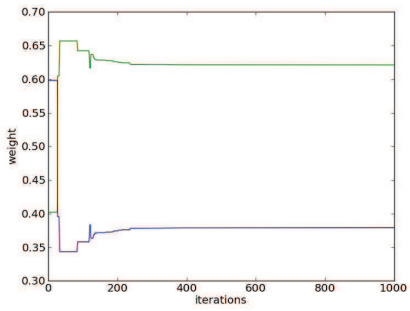


(d) support points at iteration 1000

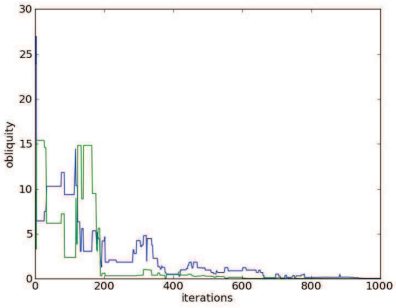
Figure 2.1: For $\#\text{supp}(\mu_i) \leq 2$, $i = 1, 2, 3$, the maximizers collapse to two-point support. Velocity and obliquity marginals each collapse to a single Dirac mass, while the plate thickness marginal collapses to have support on the extremes of its range. Note the perhaps surprising result that the probability of non-perforation is maximized by a distribution supported on the minimal, not maximal, impact obliquity.



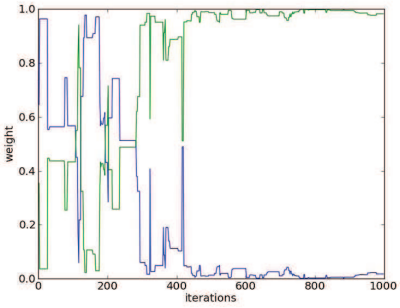
(a) convergence for thickness



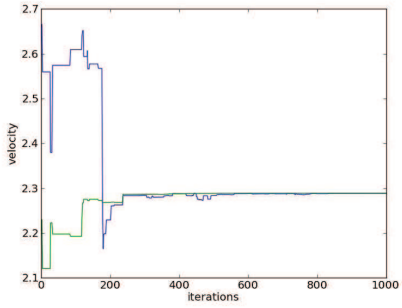
(b) convergence for thickness weight



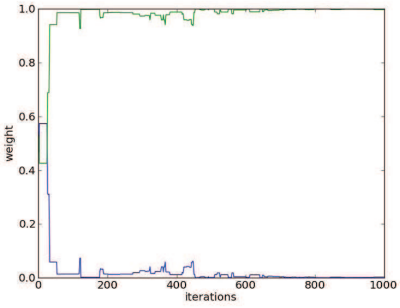
(c) convergence for obliquity



(d) convergence for obliquity weight



(e) convergence for velocity



(f) convergence for velocity weight

Figure 2.2: Time evolution of the maximizers for $\#\mathrm{supp}(\mu_i) \leq 2$ for $i = 1, 2, 3$, as optimized by *mystic*. Thickness quickly converges to the extremes of its range, with a weight of 0.621 at 60 mils and a weight of 0.379 at 105 mils. The degeneracy in obliquity at 0 causes the fluctuations seen in the convergence of obliquity weight. Similarly, velocity converges to a single support point at 2.289 $\mathrm{km} \cdot \mathrm{s}^{-1}$, the ballistic limit velocity for thickness 105 mils and obliquity 0 (see (2.2)).

control the amount of mutation that occurs at each iteration: `CrossProbability` and `ScalingFactor`. `CrossProbability` is the chance that a mutation will occur, while `ScalingFactor` is the relative size of the resulting mutation.

For the problem at hand, the optimizer is configured to use a trial population size of `npop = 40`, `CrossProbability = 0.9`, `ScalingFactor = 0.9`, and mutation strategy = `Best1Exp` [1]. Hence, there are forty trial solutions generated each iteration, where a `trialSolution` is equal to the current `bestSolution` plus the scaled difference of two of the other randomly selected trial solution `candidates`. The trial solution is altered (*i.e.* "undergoes a mutation") 90% of the time. The optimizer begins by selecting initial parameter values at random from a uniform distribution over the bounds, and the following code demonstrates how the strategy is used to generate a new parameter value at each successive iteration.

```
if random.random() >= CrossProbability:
    trialSolution = bestSolution
else:
    trialSolution = bestSolution+ScalingFactor*(candidate1 - candidate2)
```

This is the standard implementation of the `Best1Exp` mutation strategy for Differential Evolution, as detailed in [3] (we also refer to [3] for a precise definition of candidate1 and candidate2).

The trial parameter set `param` is the collection of `trialSolutions`. After generating `param`, *mystic* evaluates the `costFunction` at each iteration to generate the goodness of fit, and then checks for convergence. For OUQ, the fit parameters `param` are the positions and weights of the 1D discrete measure support along each axis of the hypercube. For example, if we use two Dirac masses as support in each direction, the fit parameters used will be:

```
param = [wx1, wx2, x1, x2, wy1, wy2, y1, y2, wz1, wz2, z1, z2]
```

Mystic provides the ability to impose both bounds constraints and non-bounds constraints in a general way. A constraint function `constrain` is applied at every optimizer iteration, forcing all random variables generated as a trial parameter set to satisfy the non-bounds constraints before the cost function is evaluated.

```
param = constrain(param)
cost = costFunction(param)
```

Bounds constraints are enforced on the random variables at each iteration by imposing an infinite potential well on the cost function, using the keyword `SetStrictRanges` [1].

`ChangeOverGeneration(tolerance=1e-4, generations=10)` was selected as the termination condition [1]. Thus convergence is defined as when the change in cost over ten iterations is less than 0.0001. If the convergence criteria are met, the `param` are returned as calculated global maximizers `opt_param`, with the resulting calculated maximum at `opt_cost = costFunction(opt_param)`. The optimizer thus provides `opt_cost` as the numerical calculation of $\mathcal{U}(\mathcal{A})$.

For the example in Section 2, the cost function is (1.2), where $h = x$, $\theta = y$, and $v = z$. Similarly for h, θ, v, the constraints imposed by `constrain` are provided by (1.3) and

$$m_1 \leq \mathbb{E}_\mu[H] \leq m_2, \text{where } [m_1, m_2] = [5.5, 7.5]\,\text{mm}^2,$$

$$\sum_{i=1}^{N_h} w_{h_i} = 1, \quad \sum_{j=1}^{N_\theta} w_{\theta_j} = 1, \quad \sum_{k=1}^{N_v} w_{v_k} = 1.$$

The global optimization is also constrained by bounds constraints provided by (2.3a) and

$$w_{h_i} \in [0,1] \text{ for each } i \in \{1, \ldots, N_h\},$$

$$w_{\theta_j} \in [0,1] \text{ for each } j \in \{1, \ldots, N_\theta\},$$

$$w_{v_k} \in [0,1] \text{ for each } k \in \{1, \ldots, N_v\}.$$

The constraint function `constrain` is used to ensure the trial parameters `param` generated by the optimizer satisfy the non-bounds constraints. The `param` are first unflattened to 3 one-dimensional discrete measures, and each measure is checked for weight normalization (*i.e.* that the total mass of the measure is 1); each measure for which the sum of the weights does not equal 1 is re-normalized to have total weight 1. The 3 one-dimensional discrete measures are then packed into a single three-dimensional product measure. The product measure is then checked for compliance with the mean constraint.

The mean perforation area constraint provides that the expectation of the product measure is in $[m-d, m+d] := [m_1, m_2]$ (where m is the target mean, and d defines some acceptable deviation from the target mean). If this constraint is not satisfied, then a new perforation area is imposed in $[m-d, m+d]$ is imposed by using the *expect* method provided by the measure object, and described below (imposing a new value for the mean preformation area is an optimization as described by the OUQ inner loop).

The product measure is then unpacked and flattened to produce the resulting trial parameters. The implementation of `constrain` is as follows:

```
x_measure, y_measure, z_measure = unflatten(param)

if sum(x_measure.weights) != 1.0:
    x_measure.normalize()
if sum(y_measure.weights) != 1.0:
    y_measure.normalize()
if sum(z_measure.weights) != 1.0:
    z_measure.normalize()

product_measure = pack((x_measure, y_measure, z_measure))

if product_measure.expect(H) > (m+d) or product_measure.expect(H) < (m-d):
    product_measure.expect = m +/- d
```

```
param = flatten(unpack(product_measure))
```

The OUQ inner loop. The reader may have noticed that satisfying the constraints provided by (1.3) can also be formulated as an optimization problem. The goal is to obtain a new set of support points that satisfy the condition $E[H] \in [m_1, m_2]$ (where $[m_1, m_2] = [m - d, m + d]$). The optimization is again performed in *mystic* using differential evolution. For the inner loop, the optimizer is configured to use a trial population size of 20, `CrossProbability = 0.9`, `ScalingFactor = 0.9`, and mutation strategy = `Best1Exp`. The initial parameter values and trial parameter values are generated as above. Bounds constraints are imposed on the random variables using `SetStrictRanges`, as above. The cost function for the inner loop is the least squared difference of $\mathbb{E}_\mu[H]$ and the target mean m.

```
product_measure = pack(unflatten(param))
cost = (product_measure.expect(H) - m)**2
```

The termination condition used for the inner loop is `VTR(tolerance = d^2)` [1], which defines convergence as having occured when the change in the cost from the last iteration is less than the prescribed `tolerance`. Therefore, if the convergence criteria are met, the resulting maximizers yield a `product_measure` where $\mathbb{E}_\mu[H] \in [m - d, m + d]$.

Acknowledgements

The authors gratefully acknowledge portions of this work supported by the Department of Energy National Nuclear Security Administration under Award Number DE-FC52-08NA28613 and by the National Science Foundation under Award Number DMR-0520547.

References

[1] M. McKerns, P. Hung, and M. Aivazis. Mystic: A simple model-independent inversion framework, 2009. http://dev.danse.us/trac/mystic.

[2] H. Owhadi, T. J. Sullivan, M. McKerns, M. Ortiz, and C. Scovel. Optimal uncertainty quantification. *SIAM Review*, Submitted, 2010. http://arxiv.org/pdf/1009.0679.

[3] K. V. Price, R. M. Storn, and J. A. Lampinen. *Differential Evolution*. Natural Computing Series. Springer-Verlag, Berlin, 2005. A practical approach to global optimization, With 1 CD-ROM (Windows, Macintosh and UNIX).

[4] R. M. Storn and K. V. Price. Differential evolution — a simple and efficient heuristic for global optimization over continuous spaces. *J. Global Optim.*, 11(4):341–359, 1997. http://dx.doi.org/10.1023/A:1008202821328.