Autoscaling API Tutorial

Harvey Weyandt

August 2019

Introduction

The purpose of this document is provide a basic user guide for the Autoscaling API V1.0. In it, we will describe how to use the API, how to implement custom autoscaling schemes to work with the API, and the critical aspects of the API design and implementation. The implementation details of the isoscaling (IS) and projected jacobian rows normalization (PJRN) autoscaling techniques, as defined in Sagliano et al (2017), are also given.

1 Using the API: A Three-Step Process

Using the API to autoscale a `Problem` instance is generally a three-step process:

1. **IMPORT** the `autoscale()` function and other desired API components from the package `autoscaling.api`. If IS or PJRN is desired, import the `IsoScaler` or `PJRNScaler` helper classes, respectively.
2. **CONFIGURE** autoscaling by instantiating an autoscaling helper class, such as `IsoScaler` or `PJRNScaler`. Depending on the helper class, this may require preloading some data, such as total jacobian information or bounds information.
3. **AUTOSCALE** the `Problem` instance by passing it, along with the autoscaling helper object, into `autoscale()`, *immediately before* the call to `run_driver()`.

Listing 1 shows how we might apply this process to an example in which we are trying to scale a problem with PJRN.

Listing 1: General example of autoscaling a problem with PJRN

```
# IMPORT autoscale() and PJRNScaler helper class
from autoscaling.api import autoscale, PJRNScaler
...
...
import openmdao.api as om
```

```python
prob = om.Problem()
...
...
prob.setup()
...
...
# CONFIGURE PJRN autoscaling helper object
with open('total_jac.pickle', 'rb') as file:
    total_jac = pickle.load(file)
with open('lower_bounds.pickle', 'rb') as file:
    lower_bounds = pickle.load(file)
with open('upper_bounds.pickle', 'rb') as file:
    upper_bounds = pickle.load(file)
autoscaler = PJRNScaler(jac, lbs, ubs)

# AUTOSCALE prob with PJRN autoscaling helper
autoscale(prob, autoscaler=autoscaler)

prob.run_driver()
```

Example: Autoscaling the "bad" brachistochrone problem with PJRN

blah

2 Extending the API: The `AutoScaler` Base Class

To implement a custom autoscaling method that is compatible with the API, one must simply create an autoscaling helper class that inherits from the API's `AutoScaler` abstract base class. Proper inheritance from `AutoScaler` is the only hard requirement for an autoscaling helper class to be compatible with the `autoscale()` method.

There are two components to the fundamental autoscaling helper interface provided by `AutoScaler` that must be defined upon inheritance:

1. **Reference value dictionaries** (`refs`, `ref0s`, `defect_refs`)
   `AutoScaler` always initializes three empty `dict`s upon construction: `refs`, `ref0s`, and `defect_refs`. These are to be defined by the user so that they map *global* (a.k.a. absolute) input/output names to their appropriate corresponding `refs`, `ref0s`, and `defect_refs`, respectively. (To be clear, global names are the kind that appear, for example, in the key pairs of the total jacobian obtained by a call to `compute_totals()`.)

2. **Initialization method** (`initialize()`)
   While it is not a hard requirement to explicitly specify anywhere the ref-

erence value dictionaries, an error *will* occur if you try to derive from `AutoScaler` without first defining the `initialize()` attribute. The purpose of this method is to perform all initialization and configuration actions necessary for the helper, such as actually setting the reference value dictionaries according to the autoscaling method being implemented. As such, the method can be defined with any number of parameters, corresponding to the data required to configure the autoscaling helper.

In short, the `autoscale()` method requires that its autoscaler helper argument define reference value dictionaries, which can be appropriately defined by deriving from `AutoScaler` and defining the `initialize()` attribute to populate its `refs`, `ref0s`, and `defect_refs` attributes.

[TODO: Rewrite, in light of the above interpretation.]

Example: Implementing "variables only" autoscaling

blah

3 API Design and Implementation Details

The `autoscale()` function

All of the actual autoscaling of a given `Problem` object is done through the API's `autoscale()` function.

Remark: Passing `None` in through the `autoscaler` keyword argument is equivalent to using no autoscaling; nothing is done to the problem.

The `AutoScaler` abstract base class

blah

Isoscaling: Definition and the `IsoScaler` helper class

Let $\mathbf{V}$ be the set of *discretized state variables*, i.e. the set of state variables that make up the NLP problem generated from the OCP via the given collocation method. Let $\mathbf{F}$ then be the set of collocation defect constraint variables generated from the OCP dynamic constraints.

What are the sizes of $\mathbf{V}$ and $\mathbf{F}$, exactly?

Since our approach here is application-oriented, for our purposes it is most instructive to think of the discretized state variable set $\mathbf{V}$ as a *dictionary*—in the Python sense—mapping a variable's *identifier* or *name string* v to its *value*, which we will denote using Python's index notation by $\mathbf{V}[v]$. We will think of similar sets in the same way; for instance, the defect set $\mathbf{F}$ will be thought of henceforth as a dictionary mapping a defect's identifier/name string f to its value $\mathbf{F}[f]$.

It is important to note that $\mathbf{F}$...

Definition 1. Isoscaling (IS) is an affine scaling scheme, applicable to OCPs transcribed via a collocation method, in which the defects are scaled in the same manner as their corresponding discretized state variables, but with no shift:

$$\hat{\mathbf{V}}[v] \equiv \mathbf{a}[v]\mathbf{V}[v] + \mathbf{b}[v]$$
$$\hat{\mathbf{F}}[f] \equiv \mathbf{a}_{\mathbf{F}}[f]\mathbf{F}[f]$$

for all discretized variable identifiers v and defect identifiers f, where $\mathbf{a}$ is the set of discretized state multipliers defined such that

$$\mathbf{a}[v] = \frac{1}{v^U - v^L}$$

where v^U, v^L denote the upper and lower bounds, respectively, for the variable identified by v; $\mathbf{b}$ is the set of discretized state shifts defined such that

$$\mathbf{b}[v] = -\frac{v^L}{v^U - v^L}$$

and $\mathbf{a}_{\mathbf{F}}$ is the subset of $\mathbf{a}$ defined such that

$$\mathbf{a}_{\mathbf{F}}[f] = \mathbf{a}[s(f)]$$

where $s(f)$ is the identifier for the discretized state variable associated with the defect identified by f. Note that $\mathbf{a}_{\mathbf{F}}$ is a *subset*, generally, because discretized states corresponding to OCP *controls* do not have corresponding defects.

(See Sagliano et al (2017) for an equivalent definition using matrices and vectors instead of sets thought of in terms of Python dictionaries.)

How exactly do we get `refs`, `ref0s`, and `defect_refs` from this definition?

Let v be a discretized state variable identifier. The `ref` value for v, given by `refs`$[v]$, is by definition the value of v that scales to 1. Thus

$$\texttt{refs}[v] = \frac{1 - \mathbf{b}[v]}{\mathbf{a}[v]} = v^U$$

The `ref0` value for v, given by `ref0s`$[v]$, is by definition the value of v that scales to 0, so

$$\texttt{ref0s}[v] = -\frac{\mathbf{b}[v]}{\mathbf{a}[v]} = v^L$$

Let f be a defect constraint variable identifier. The `defect_ref` value for f, given by `defect_refs`$[f]$, is by definition the value of f that scales to 1, so

$$
\begin{aligned}
\texttt{defect\_refs}[f] &= \frac{1}{\mathbf{a}_{\mathbf{F}}[f]} \\
&= \frac{1}{\mathbf{a}[s(f)]} \\
&= s(f)^U - s(f)^L
\end{aligned}
$$

These are precisely the definitions that `IsoScaler` helper class instances use to populate their `refs`, `ref0s`, and `defect_refs` dictionaries.

[TODO: Add stuff about IsoScaler helper class]

PJRN: Definition and the `PJRNScaler` helper class

Sagliano et al (2017) uses matrices and vectors to define the projected jacobian rows normalization (PJRN) scaling technique essentially as follows:

Let $\mathbf{V}, \mathbf{F}, \mathbf{G}$ be the vector of discretized states, the vector of defect constraint variables, and the vector of path constraint variables, respectively. Their scaled counterparts $\hat{\mathbf{V}}, \hat{\mathbf{F}}, \hat{\mathbf{G}}$ under **projected jacobian rows normalization** (PJRN) are given by

$$
\begin{aligned}
\hat{\mathbf{V}} &= \mathrm{K}_{\mathbf{V}}\mathbf{V} + \mathrm{b} \\
\hat{\mathbf{F}} &= \mathrm{K}_{\mathbf{F}}\mathbf{F} \\
\hat{\mathbf{G}} &= \mathrm{K}_{\mathbf{G}}\mathbf{G}
\end{aligned}
$$

where $\mathrm{K}_{\mathbf{V}}, \mathrm{b}$ are defined as in isoscaling, $\mathrm{K}_{\mathbf{F}}$ is an $n_s n \times n_s n$ diagonal matrix whose diagonal entries are given by

$$\mathrm{ent}_{i,i}\mathrm{K}_{\mathbf{F}} = \frac{1}{|\nabla\mathbf{F} \cdot \mathrm{K}_{\mathbf{V}}^{-1}|_i}$$

and $\mathrm{K}_{\mathbf{G}}$ is an $n_g n \times n_g n$ diagonal matrix whose diagonal entries are given by

$$\mathrm{ent}_{i,i}\mathrm{K}_{\mathbf{G}} = \frac{1}{|\nabla\mathbf{G} \cdot \mathrm{K}_{\mathbf{V}}^{-1}|_i}$$

The notation $|\cdot|_i$ is understood to mean the norm of row i:

$$|\cdot|_i \equiv ||\mathrm{row}_i(\cdot)||$$

How do these equations translate into OpenMDAO code? Specifically, how are reference values to be computed?

Since the K matrices are diagonal, we can write the scaling equations as

$$\begin{aligned}
\hat{\mathbf{V}}_i &= \mathrm{ent}_{i,i}\mathrm{K}_{\mathbf{V}} \cdot \mathbf{V}_i + \mathrm{b}_i & (i = 1, \ldots, N) \\
\hat{\mathbf{F}}_i &= \mathrm{ent}_{i,i}\mathrm{K}_{\mathbf{F}} \cdot \mathbf{F}_i & (i = 1, \ldots, n_s n) \\
\hat{\mathbf{G}}_i &= \mathrm{ent}_{i,i}\mathrm{K}_{\mathbf{G}} \cdot \mathbf{G}_i & (i = 1, \ldots, n_g n)
\end{aligned}$$

From these it is easier to derive expressions for the desired reference values.

The reference values for the discretized state variables $\mathbf{V}$ are the same as in isoscaling; what's different about this method is what happens to the constraints.

Let f be a defect constraint variable identifier, and let $i = i(f)$ be the index of $\mathbf{F}$ corresponding to the value of f. By definition, the `defect_ref` for f is the value of f that scales to 1. Thus, if the `defect_ref` for f is given by `defect_ref`(f), then

$$1 = \mathrm{ent}_{i,i}\mathrm{K}_{\mathbf{F}} \cdot \texttt{defect\_ref}(f)$$

This can be solved for `defect_ref`(f):

$$\begin{aligned}
\texttt{defect_ref}(f) &= \frac{1}{\mathrm{ent}_{i,i}\mathrm{K}_{\mathbf{F}}} \\
&= |\nabla\mathbf{F} \cdot \mathrm{K}_{\mathbf{V}}^{-1}|_i \\
&= ||\mathrm{row}_i(\nabla\mathbf{F} \cdot \mathrm{K}_{\mathbf{V}}^{-1})|| \\
&= ||\mathrm{row}_i\nabla\mathbf{F} \cdot \mathrm{K}_{\mathbf{V}}^{-1}|| \\
&= ||\mathrm{row}_i\frac{\partial\mathbf{F}}{\partial\mathbf{V}} \cdot \mathrm{K}_{\mathbf{V}}^{-1}|| \\
&= ||\frac{\partial\mathbf{F}_i}{\partial\mathbf{V}} \cdot \mathrm{K}_{\mathbf{V}}^{-1}|| \\
&= ||\Big[\frac{\partial\mathbf{F}_i}{\partial\mathbf{V}_j} \cdot \mathrm{ent}_{j,j}\mathrm{K}_{\mathbf{V}}^{-1}\Big]_{j=1,\ldots,N}|| \\
&= ||\Big[\frac{\partial\mathbf{F}_i}{\partial\mathbf{V}_j} \cdot \mathrm{rng}(\mathbf{V}_j)\Big]_{j=1,\ldots,N}|| \\
&= \Big\{\sum_{\mathrm{v}\in\mathbf{V}}\Big(\frac{\partial\mathbf{F}_i}{\partial\mathrm{v}} \cdot \mathrm{rng}(\mathrm{v})\Big)^2\Big\}^{1/2} \\
&= \Big\{\sum_{\mathrm{var}}\sum_{\mathrm{node}}\Big(\frac{\partial\mathbf{F}_i}{\partial\mathrm{v}(\mathrm{var},\mathrm{node})} \cdot \mathrm{rng}(\mathrm{var})\Big)^2\Big\}^{1/2} \\
&= \Big\{\sum_{\mathrm{var}}\Big[\mathrm{rng}^2(\mathrm{var})\sum_{\mathrm{node}}\Big(\frac{\partial\mathbf{F}_i}{\partial\mathrm{v}(\mathrm{var},\mathrm{node})}\Big)^2\Big]\Big\}^{1/2} \\
&= \Big\{\sum_{\mathrm{var}}\Big[\mathrm{rng}^2(\mathrm{var})\sum_{\mathrm{node}}\Big(\texttt{Jac[fvar,var][fnode][node]}\Big)^2\Big]\Big\}^{1/2}
\end{aligned}$$

where `rng` denotes the range (max minus min) of the argument; `Jac` is the dictionary of total jacobian information—as can be obtained from `compute_totals()`; `var` sums over the identifiers of the *continuous* problem inputs; `node` sums over the particular node indices for each `var`; `fvar` is the identifier of the continuous problem state to which the defect f corresponds; and `fnode` is the particular node index of f. It is precisely this equation that manifests itself in the `PJRNScaler` helper class definition.

This same derivation can be used for the `ref` values of the path constraints. Let g be a path constraint variable identifier, and let $i = i(g)$ be the index of $\mathbf{G}$ corresponding to the value of g. Then

$$\texttt{ref}(g) = \left\{ \sum_{\text{var}} \left[\text{rng}^2(\text{var}) \sum_{\text{node}} \Big(\texttt{Jac[gvar,var][gnode][node]} \Big)^2 \right] \right\}^{1/2}$$

Note that $\texttt{ref0}(g) = 0$ for all g, since $\mathbf{G}$ is scaled *linearly*.