A Primer on Scaling for Multi-Disciplinary Optimization in OpenMDAO and Dymos

Harvey Weyandt

Contents

1 Scaling Fundamentals **2**
What scaling is and why we do it 2
The common scaling transformations 3
Rules of thumb and words of warning 4

2 Scaling in OpenMDAO **6**
Scalers and adders (deprecated) 6
Reference values 6
A Simple Example 7
Under the hood 10

3 Scaling in Dymos **11**
Defect constraint reference values 11
An example of defect scaling 11
Under the hood 11

4 Selected Scaling Techniques **12**
Isoscaling 12
Projected Jacobian Rows Normalization 14

References, Resources, External Links **16**

Equations Cheatsheet **17**

FAQ **18**

1 Scaling Fundamentals

What scaling is and why we do it

A typical issue plaguing most treatments of the topic of scaling is a vagueness of terms, so to begin our treatment of scaling here we will clarify some terminology.

Problem scaling—or simply **scaling**, when understood to be with respect to a problem—is the act of transforming one optimization problem into another by applying transformations to its variables and rewriting the problem in terms of the transformed variables. Note that the variables can be inputs or outputs.

The problem transformation applied in the act of problem scaling is called a **problem scaling transformation**. The individual transformations applied to the problem variables during scaling are called **variable scaling transformations**, and the act of applying those is called **variable scaling**—or just **scaling**, when understood to be with respect to variables, not an entire problem. Scaling transformations of problems and variables can simply be called **scaling transformations** when the context is clear. The inverses of scaling transformations, when they are available, are called **descaling transformations**, and applying such transformations to a scaled problem or its variables is called **descaling**.

In the context of scaling one problem P into a new problem $\hat{P}$, the original problem P is called the **unscaled problem** while the new problem $\hat{P}$ is called the **scaled problem**. The variables of these problems are then dubbed "unscaled" (a.k.a. "physical") and "scaled", respectively.

Suppose we wanted to numerically solve a problem P with some solution method implementation `M` that is given initial guess information P^*_{guess}. The convergence behavior we would observe while `M` runs is determined in part by the formulation of P. For example, if some problem variables are evaluated in units that are too large or too small relative to those of other variables, then operations between these variables may incur significant error in the numerical results, to such a degree that convergence is either egregiously slow, egregiously inaccurate, or impossible altogether.

The convergence issues in the above example may be circumvented by an informed (or lucky) choice of `M` and P^*_{guess}, but not always. Thus the best course of action in this case, if `M` and P^*_{guess} are not to be questioned, is to reformulate P in terms of new units, in such a way that the errors occurring in the execution of `M` are minimal and consequently the convergence behavior of `M` is optimal. Since changing the units of a variable amounts to applying a transformation to that variable, and problem scaling is precisely the application of transformations like these to the problem variables, this is another way of saying that the best course of action is to scale P into a new problem $\hat{P}$ (particularly via changes in units). After applying the scaling transformation, $\hat{P}$ can then be solved to yield some optimum $\hat{x}^*$, which can be descaled to an optimum x^* of P by converting back to the old units.

Therein lies the utility of scaling, in general. If we choose our scaling transformation just right, we can transform a "poorly-scaled" problem P, conducive of poor convergence, into a "well-scaled" problem $\hat{P}$, conducive of more desirable convergence (with respect to a given solution method implementation and some given initial guess information). The scaled solution to $\hat{P}$ can then be descaled to, hopefully, yield a close approximation to the unscaled solution of P.

In summary, scaling provides an *indirect* method for solving optimization problems. The method consists of the following three steps:

1. *Scale* problem P into problem $\hat{P}$
2. *Solve* $\hat{P}$ directly for scaled optimum $\hat{x}^*$
3. *Descale* $\hat{x}^*$ to approximate unscaled optimum x^*

The nice thing is that OpenMDAO does all of this automatically! All you need to do is tell it what scaling transformation to apply. This you do by specifying units, if desired, and setting reference values. (See Section ?? for details.)

The common scaling transformations

Recall that a problem scaling transformation S transforms a problem P into a new problem $\hat{P}$:

$$S(P) = \hat{P}$$

It does this by applying a set of variable scaling transformations to the variables of P. For each variable v of P, the problem scaling transformation S provides a variable scaling transformation S_v to transform it into scaled variable $\hat{v}$ of $\hat{P}$:

$$S_v(v) = \hat{v}$$

If a variable v is untouched by S, then S_v is simply the identity transformation, so that $\hat{v} = S_v(v) = v$.

While scaling transformations are defined above in a very general sense, specific types of transformations are most commonly used. The most significant and commonplace types are "linear" and "affine", as described below.

Linear Scaling

S is called **linear** if the variable transformations it defines are all linear maps—that is, if

$$\hat{v} = S_v(v) = \alpha_v v$$

for all v, where α_v is constant.

The majority of unit changes encountered in scientific work are linear scaling transformations. For example, if our problem had a scalar state variable x

expressed in meters (m), then in units of millimeters (mm) it is 10^3x, so the change from m to mm corresponds to a linear scaling transformation

$$\hat{x} = S_x(x) = 10^3x$$

Affine Scaling

Generalizing linear scaling to include possible "shifts" in variables leads to the notion of affine scaling. S is called **affine** (pronounced "ay-fine") if the variable transformations it defines are all affine maps—that is, if

$$\hat{v} = S_v(v) = \alpha_v v + \beta_v$$

for all v, where α_v,β_v are constant. Note that if all $\beta_v = 0$, that means that S is linear. All linear transformations are affine, but not all affine transformations are linear.

Some unit conversions require affine transformations. For instance, converting a variable T measured in degrees Celsius (°C) to a new variable $\hat{T}$ measured in degrees Fahrenheit (°F) requires the affine scaling transformation

$$\hat{T} = S_T(T) = \frac{9}{5}T + 32$$

More will be said about linear and affine scaling later, since those are the types of transformations that the OpenMDAO user interface currently supports. (See Section ?? for details.)

Rules of thumb and words of warning

How do you know when a problem is poorly-scaled? How do you know that scaling the problem can yield sufficiently significant convergence benefits? And how do you know what scaling transformations to use to reap those benefits? These are all very difficult, nuanced questions to answer precisely, and various aspects of these questions are still open areas of study, but nonetheless a few rules of thumb have been observed to work well in most instances.

Firstly, a problem may be poorly-scaled if at least one of its variables is allowed to take on values spanning several orders of magnitude. A distance measure that can assume significant values 0.01m and 10000m, for example, spans at least six orders of magnitude and is likely to cause poor scaling in a problem. Outliers can have the same effect, stretching the precision with which a variable must be measured, thus causing it to span greater orders of magnitude, leading to poor scaling.

Secondly, a problem may be poorly-scaled if its variables can take on values that span several orders of magnitude *relative to each other.* This phenomenon has the potential to disturb gradient-based methods because the optimization may consequently be too sensitive to changes in some variables while not being sensitive enough to changes in others. Imagine the contour graph of an objective function. If one variable is of, say, a far greater order of magnitude than the others, then the contours will be excessively "stretched" along that variable's coordinate axis, eliminating much of the curvature necessary for good convergence behavior under gradient-based methods.

Usually it helps to scale problem variables to be less disparate with respect to themselves and each other, as described above. The conventional way to do this is to "normalize" each variable so that its scaled value lies within $[0, 1]$ or $[-1, 1]$. Ross et al (2018) warn, however, that this is not an end-all-be-all solution to poor scaling, because normalizing the *primal variables* in this way may have adverse effects on the scales of the *dual variables*, consequently disturbing the "balancing" of the problem with regards to the necessary conditions for optimality generated by Pontryagin's principle. Still, it is often worth trying.

2 Scaling in OpenMDAO

The OpenMDAO user interface provides support for linear and affine scaling, both *indirectly* by allowing users to specify units for variables of interest, and *directly* via either `scaler/adder` or `ref/ref0` keyword argument assignments.

Warning: As of OpenMDAO V2, the `scaler/adder` interface for direct scaling is deprecated, so use `ref/ref0` instead! The following discussions of `scaler` and `adder` are only made to explain how to interpret the older programs that use them.

Scalers and adders (deprecated)

To scale a variable this way, you must specify a `scaler` and an `adder`. How do these values translate to an affine scaling transformation? Well, if x is the unscaled variable, then its scaled counterpart $\hat{x}$ is given by

$$\hat{x} = \texttt{scaler} \cdot (x + \texttt{adder}) \tag{1}$$

This means that, in terms of an affine transformation, `scaler` is the constant multiplier and $\texttt{scaler} \cdot \texttt{adder}$ is the constant shift term.

Reference values

The preferred way to directly scale variables is through *reference values*. Most variables defined in OpenMDAO are associated with at least two types of reference values: `ref` and `ref0`.

The `ref` value for a variable is defined to be the value that scales to 1, and the `ref0` value is that which scales to 0. Symbolically,

$$\begin{aligned} \texttt{ref} &\mapsto 1 \\ \texttt{ref0} &\mapsto 0 \end{aligned}$$

Hence the scaling transformation from the corresponding variable x to $\hat{x}$ is precisely

$$\hat{x} = \frac{x - \texttt{ref0}}{\texttt{ref} - \texttt{ref0}} \tag{2}$$

This means that, in terms of an affine transformation, the constant multiplier is $\frac{1}{\texttt{ref}-\texttt{ref0}}$ and the constant shift term is $-\frac{\texttt{ref0}}{\texttt{ref}-\texttt{ref0}}$.

Note that, given variable x, if x^U denotes an upper bound of x and x^L denotes a lower bound of x, then the domain interval $[x^L, x^U]$ is transformed as follows:

$$\begin{aligned}[x^L, x^U] &\mapsto [\hat{x}^L, \hat{x}^U] \\ &= \left[\frac{x^L - \texttt{ref0}}{\texttt{ref} - \texttt{ref0}}, \frac{x^U - \texttt{ref0}}{\texttt{ref} - \texttt{ref0}}\right]\end{aligned}$$

Problem: Find `ref`,`ref0` such that the resulting scaling transformation maps the physical domain interval $[a, b]$ to the interval $[\hat{a}, \hat{b}]$.

Solution: Equation (2) gives $\hat{a}, \hat{b}$ in terms of $a, b, \texttt{ref}, \texttt{ref0}$, and solving the resulting linear system for `ref`,`ref0` yields

$$\begin{aligned}\texttt{ref} &= \frac{1 - \hat{b}}{\hat{a} - \hat{b}} a + \frac{\hat{a} - 1}{\hat{a} - \hat{b}} b \\ \texttt{ref0} &= -\frac{\hat{b}}{\hat{a} - \hat{b}} a + \frac{\hat{a}}{\hat{a} - \hat{b}} b\end{aligned}$$

Note that `ref`,`ref0` are simply linear combinations of the physical bounds whose coefficients are determined by the scaled bounds.

Problem: Given `scaler` and `adder`, compute `ref` and `ref0`.

Solution: Since `ref` is defined as the physical value that scales to 1, and `ref0` is defined as the physical value that scales to 0, Equation (1) gives equations that can be solved for `ref` and `ref0` to yield the following identities:

$$\begin{aligned}\texttt{ref} &= \frac{1}{\texttt{scaler}} - \texttt{adder} \\ \texttt{ref0} &= -\texttt{adder}\end{aligned}$$

If instead `scaler` and `adder` were desired from `ref` and `ref0`, the above equations would form a linear system that could be solved for those values to yield the following inverse identities:

$$\begin{aligned}\texttt{scaler} &= \frac{1}{\texttt{ref} - \texttt{ref0}} \\ \texttt{adder} &= -\texttt{ref0}\end{aligned}$$

Note that values for `ref`/`ref0`, if they are set, override values for `scaler`/`adder` in OpenMDAO code, since the latter are deprecated.

A Simple Example

Suppose we wish to solve the following simple optimization problem in OpenMDAO:

$$\begin{aligned} \text{minimize} \quad & f \equiv 10^6 x + 7y \\ \text{subject to} \quad & g \equiv 10^7 x + 8y \leq 500 \\ & h \equiv 10^{12} x + 10^6 y \geq 50 \cdot 10^6 \end{aligned}$$

Listing 1 shows an implementation of this problem in OpenMDAO. Note that Scipy's SLSQP is designated as the optimizer and our initial guess is $(x, y) = (0, 1)$.

Listing 1: Simple Optimization Problem Example in OpenMDAO

```python
import openmdao.api as om

prob = om.Problem()
model = prob.model

ivc = model.add_subsystem('ivc', om.IndepVarComp())
ivc.add_output('x')
ivc.add_output('y')

model.add_subsystem('obj', om.ExecComp('f = 1e6*x + 7*y'))
model.add_subsystem('con1', om.ExecComp('g = 1e7*x + 8*y'))
model.add_subsystem('con2', om.ExecComp('h = 1e12*x + 1e6*y'))

model.connect('ivc.x', ['obj.x', 'con1.x', 'con2.x'])
model.connect('ivc.y', ['obj.y', 'con1.y', 'con2.y'])

prob.driver = om.ScipyOptimizeDriver()
prob.driver.options['optimizer'] = 'SLSQP'

model.add_design_var('ivc.x')
model.add_design_var('ivc.y')
model.add_objective('obj.f')
model.add_constraint('con1.g', upper=500)
model.add_constraint('con2.h', lower=5e7)

prob.setup()

prob.set_val('ivc.x', 0)
prob.set_val('ivc.y', 1)

prob.run_driver()
```

Running this implementation as-is will not yield a solution, due to numerical difficulties. (To be sure, using a more powerful optimizer like pyOptSparse's

SNOPT would overcome these difficulties in this case, but it won't always! So, for the purposes of illustrating scaling here, we will stick with SLSQP.) Luckily, the numerical difficulties can be overcome with effective problem scaling.

How do we know that scaling can help? It turns out that our example problem is poorly-scaled since convergence is generally hindered—if not outright precluded—by numerical difficulties that arise due to the vast disparity in relative orders of magnitude between the quantities x, y, f, g, h. The hint to this fact lies in the disparity between coefficients in the problem statement (some 10^6s here, an 8 here, a 10^{12} there...) For example, if we take $x = 1$ and $y = 1$, then f is greater than x, y by 6 orders of magnitude, g is greater than x, y by 7 orders of magnitude, and h is greater than x, y by a whole 12 orders of magnitude. Recall that disparity in orders of magnitude is not always an indicator of poor scaling, but here it is because the scaling of the variables can be shown to lead to a very ill-conditioned jacobian matrix, which of course causes problems many of OpenMDAO's gradient-based solution methods.

How do we approach scaling this problem to make it more well-behaved? In other words, what problem variables should we scale, and how, to make this problem well-scaled? How, then, do we set the appropriate reference values for each variable?

Perhaps the best approach in this case is to scale x to $\hat{x} = 10^6 x$ and h to $\hat{h} = 10^{-6} h$, so that our problem becomes

$$\begin{aligned} \text{minimize} \quad & f \equiv \hat{x} + 7y \\ \text{subject to} \quad & g \equiv 10\hat{x} + 8y \leq 500 \\ & \hat{h} \equiv \hat{x} + y \geq 50 \end{aligned}$$

Now our problem variables are all within three or four orders of magnitude of each other, thus making for a more well-conditioned jacobian which is much more conducive to well-behaved convergence.

Note that the idea in this example was essentially to scale variables to eliminate unwieldy coefficients. The fact that x was amplified by at least an order of 6 in each equation/inequality was a hint that x is too "small" of a variable, needing to be "scaled up" at least to $\hat{x} = 10^6 x$; and the fact that both sides of the inequality constraint on h could be reduced by an order of 6 after that was a hint that h is too "large", needing to be "scaled down" at least to $\hat{h} = 10^{-6} h$.

To apply our scaling scheme to the OpenMDAO implementation, we would need to set the `ref`/`ref0` values for x and h. (Since y, f, g are untouched, we do nothing to them.) What are these?

The `ref0` for each variable is, by definition, the value that scales to 0, which is obviously 0 for both x and h. On the other hand, the `ref` for each variable is, by definition, the value that scales to 1; thus the `ref` for x would be 10^{-6} and the `ref` for h would be 10^6.

To actually set the `ref`/`ref0` values in the code, we need to pass them in through the corresponding keyword arguments in `add_design_var()` and `add_constraint()`:

```
model.add_design_var('ivc.x', ref=1e-6)
...
model.add_constraint('con2.h', lower=5e7, ref=1e6)
```

Run the modified code and, *voila*, the optimization converges without a hitch!

Under the hood

See the Scaling section of the OpenMDAO V2 documentation for more on scaling in OpenMDAO. For more in-depth information on how OpenMDAO handles scaling, visit the Scaling Variables in OpenMDAO section.

3 Scaling in Dymos

All of the advice from the previous section on OpenMDAO still applies, since Dymos is build on top of OpenMDAO. However, there are some particular facets to Dymos that are owed consideration.

Defect constraint reference values

Dymos utilizes transcription methods, such as high-order Gauss-Lobatto and Radau pseudospectral methods, to convert dynamically-constrained optimal control problems (OCPs) into nonlinear programming (NLP) problems that are solvable within the OpenMDAO framework. The NLP problems generated by such transcription methods involve collocation defect constraints for each discretized input. These defect constraints can be scaled individually through the Dymos UI via *defect constraint reference values*, or `defect_ref` values.

[TODO: How to set defect refs]

An example of defect scaling

[TODO: Setting defect refs in X57 MPT Mod2 Power Phase]

Under the hood

Refer to the Dymos documentation for more details on scaling in Dymos.

4 Selected Scaling Techniques

[TODO: Preamble]

Isoscaling

Formulation

The first and simplest of the methods described by Sagliano et al (2017) for scaling dynamically-constrained optimal control problems that are solved via transcription is the **isoscaling** (IS) method. In IS, the discretized states/-controls $\mathbf{V}$ and their corresponding collocation defect constraints $\mathbf{F}$ are scaled according to

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

where K is a diagonal matrix whose diagonal elements are given by

$$\mathrm{ent}_{i,i}\mathrm{K} = \frac{1}{\mathbf{V}_i^U - \mathbf{V}_i^L}$$

where $\mathbf{V}_i^U, \mathbf{V}_i^L$ are the upper and lower bounds of $\mathbf{V}_i$, respectively; $\mathrm{K}_{\mathbf{F}}$ is the diagonal submatrix of K formed by taking only the rows and columns corresponding to discretized *states*, not controls; and b is a vector whose elements are given by

$$\mathrm{b}_i = \frac{\mathbf{V}_i^L}{\mathbf{V}_i^U - \mathbf{V}_i^L}$$

To be perfectly explicit, the objects $\mathbf{V}, \hat{\mathbf{V}}, \mathbf{V}^U, \mathbf{V}^L, \mathbf{F}, \hat{\mathbf{F}}, \mathrm{K}, \mathrm{K}_{\mathbf{F}}, \mathrm{b}$ have the following dimensions:

$$\begin{aligned}\dim\mathbf{V} = \dim\hat{\mathbf{V}} = \dim\mathbf{V}^U = \dim\mathbf{V}^L &= \dim\mathrm{b} = N \times 1 \\ \dim\mathbf{F} = \dim\hat{\mathbf{F}} &= n_s n \times 1 \\ \dim\mathrm{K} &= N \times N \\ \dim\mathrm{K}_{\mathbf{F}} &= n_s n \times n_s n\end{aligned}$$

where N is the number of state variables in the *discrete* NLP problem resulting from transcription, n_s is the number of state variables in the *continuous* optimal control problem, and n is the number of discretization nodes utilized in the transcription.

Implementation

How exactly do the equations defining IS translate into reference values that can be used in OpenMDAO/Dymos?

[Since $\mathbf{F}$ is a matrix with $n_s n$ rows, we may choose to index rows not with a single integer index i, but rather with one string index v and another integer index μ, where v specifies the name of a discretized variable and μ specifies the particular node of that variable's discretization. This choice enables a straightforward interpretation of the formulae in terms of `refs`, `ref0`s, and `defect_refs`.]

The `defect_ref` to be set for state x, for all discretization nodes, is given by

$$\texttt{defect\_ref}_x = x^U - x^L$$

i.e. the range of x. (Recall that x^U,x^L are to be chosen artificially if x is unbounded.)

As for the states (and controls) themselves, `ref` and `ref0` for each state/-control v are given simply by the upper and lower bounds v^U, v^L, respectively:

$$\texttt{ref}_v = v^U$$
$$\texttt{ref0}_v = v^L$$

Properties, pros and cons

Note that the formulation of IS given above does not touch path constraints $\mathbf{G}$. This is intentional, though Sagliano et al do not explain the rationale. Likely, the path constraints $\mathbf{G}$—and, indeed, non-defect constraints in general—are not scaled by K because such constraints do not depend upon the states/controls, in the way that the defect constraints do. For defect constraints, the constraint variable and the constraint itself are both dependent upon the state/control to which it pertains; whereas for non-defect constraints, the constraint variable is obviously going to be dependent on *some* subset of the states/controls, while the constraint *itself*—the equality or inequality statement relating the constraint variable to some fixed value(s)—are *not* dependent on any such subset.

An obvious benefit to IS is its simplicity. Of all the ways to scale dynamically-constrained optimal control problems subject to transcription in their solution procedures, IS is on the lower end of the spectrum regarding the number of required computations. Another benefit is that very little information is required to perform the relevant computations. Particularly, IS only needs upper and lower bounds on the discretized variables. Therefore, IS requires no prior solution attempts, making it a good "first resort" for trying to autoscale an ill-behaved problem.

However, since IS does not scale non-defect constraints, it is not a method to be used alone, but rather a method to be used in tandem with some other manual or automatic scaling scheme that takes care of such constraints. Also, according to Sagliano, IS fails to take into account the jacobian information describing the dynamic relationships between states/controls and their defects, not to mention the character of the transcription method itself. This negligence, of course, may result in an inadvertently ill-conditioned jacobian—or, at least, one that is not conditioned nearly as well as it could be.

Projected Jacobian Rows Normalization

Formulation

The aforementioned drawbacks to isoscaling (IS) are significantly ameliorated by the design of a more complex autoscaling method called **projected jacobian rows normalization** (PJRN). Introduced by Sagliano et al (2017), PJRN scales not only discretized states/controls $\mathbf{V}$ and their associated defects $\mathbf{F}$, but also the path constraints $\mathbf{G}$, according to

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

where $\mathrm{K}_{\mathbf{V}}, \mathrm{b}$ are the same as K, b, respectively, in IS, and $\mathrm{K}_{\mathbf{F}}, \mathrm{K}_{\mathbf{G}}$ are diagonal matrices whose diagonal elements are given by

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

where $|*|_i$ denotes the (euclidean) norm of the ith row of $*$, and $\nabla\mathbf{F}$,$\nabla\mathbf{G}$ are taken with respect to $\mathbf{V}$ and are thus subjacobians of a total jacobian. (Which total jacobian, exactly? More on that in the Properties section.)

To be perfectly explicit, the objects $\mathbf{V}, \hat{\mathbf{V}}, \mathbf{V}^U, \mathbf{V}^L, \mathbf{F}, \hat{\mathbf{F}}, \mathbf{G}, \hat{\mathbf{G}}, \nabla\mathbf{F}, \nabla\mathbf{G}, \mathrm{K}_{\mathbf{V}}, \mathrm{K}_{\mathbf{F}}, \mathrm{K}_{\mathbf{G}}, \mathrm{b}$ have the following dimensions:

$$\begin{aligned}\dim\mathbf{V} = \dim\hat{\mathbf{V}} = \dim\mathbf{V}^U = \dim\mathbf{V}^L = \dim\mathrm{b} &= N\times 1\\ \dim\mathbf{F} = \dim\hat{\mathbf{F}} &= n_s n\times 1\\ \dim\mathbf{G} = \dim\hat{\mathbf{G}} &= n_g n\times 1\\ \dim\mathrm{K}_{\mathbf{V}} &= N\times N\\ \dim\mathrm{K}_{\mathbf{F}} &= n_s n\times n_s n\\ \dim\mathrm{K}_{\mathbf{G}} &= n_g n\times n_g n\\ \dim\nabla\mathbf{F} &= n_s n\times N\\ \dim\nabla\mathbf{G} &= n_g n\times N\end{aligned}$$

where N is the number of state variables in the *discrete* NLP problem resulting from transcription, n_s is the number of state variables in the *continuous* optimal control problem, n_g is the number of path constraints in the optimal control problem, and n is the number of discretization nodes utilized in the transcription.

Implementation

Let `Jac` be a valid total jacobian matrix for the problem—e.g., one returned by calling `compute_totals()` on the problem's representative `Problem` instance. `Jac` is essentially a dictionary mapping a pair of keys to a rectangular array of numbers. The individual elements of these arrays that are *pertinent to the computation of* $\nabla \mathbf{F}$ are accessed by the following general scheme:

```
Jac[fname,vname][fnode][vnode]
```

where `fname` is the global name of a defect, `vname` is the global name of a state/control variable, and `fnode`, `vnode` are non-negative integers referencing the particular discretization nodes at which the defect and variable, respectively, are evaluated. Similarly, elements *pertinent to the computation of* $\nabla \mathbf{G}$ are given by

```
Jac[gname,vname][gnode][vnode]
```

where `gname` and `vname` are defined analogously.

Properties, pros and cons

[TODO]

Modifications

[TODO]

References, Resources, External Links

Equations Cheatsheet

FAQ