gpkit Documentation

Release 0.4.0

MIT Department of Aeronautics and Astronautics

May 18, 2016

GPkit is a Python package for defining and manipulating geometric programming (GP) models, abstracting away the backend solver. Supported solvers are MOSEK and CVXOPT.

CHAPTER 1

Geometric Programming 101

1.1 What is a GP?

A Geometric Program (GP) is a type of non-linear optimization problem whose objective and constraints have a particular form. The decision variables in a GP must have strictly positive values (that is, they can't be zero).

GP objectives and inequalities are formed out of *monomials* and *posynomials*. In the context of GP, a monomial is defined as:

$$f(x) = c x_1^{a_1} x_2^{a_2} ... x_n^{a_n}$$

where c is a positive constant, $x_{1..n}$ are decision variables, and $a_{1..n}$ are real exponents. For example, taking x, y and z to be positive variables, the expressions

$$7x \qquad 4xy^2z \qquad \frac{2x}{y^2 z^{0.3}} \qquad \sqrt{2xy}$$

are all monomials. Building on this, a posynomial is defined as a sum of monomials:

$$g(x) = \sum_{k=1}^{K} c_k x_1^{a_1 k} x_2^{a_2 k} ... x_n^{a_n k}$$

For example, the expressions

$$x^2 + 2xy + 1 \qquad 7xy + 0.4(yz)^{-1/3} \qquad 0.56 + \frac{x^{0.7}}{yz}$$

are all posynomials. Alternatively, monomials can be defined as the subset of posynomials having only one term. Using f_i to represent a monomial and g_i to represent a posynomial, a GP in standard form is written as:

$$\begin{array}{ll} \text{minimize} & g_0(x) \\ \text{subject to} & f_i(x) = 1, \quad i = 1, ..., m \\ & g_i(x) \leq 1, \quad i = 1, ..., n \end{array}$$

Boyd et. al. give the following example of a GP in standard form:

$$\begin{array}{ll} \text{minimize} & x^{-1}y^{-1/2}z^{-1} + 2.3xz + 4xyz \\ \text{subject to} & (1/3)x^{-2}y^{-2} + (4/3)y^{1/2}z^{-1} \leq 1 \\ & x + 2y + 3z \leq 1 \\ & (1/2)xy = 1 \end{array}$$

1.2 Why are GPs special?

Geometric programs have several powerful properties:

1. Unlike most non-linear optimization problems, large GPs can be **solved extremely quickly**.
2. If there exists an optimal solution to a GP, it is guaranteed to be **globally optimal**.
3. Modern GP solvers require **no initial guesses** or tuning of solver parameters.

These properties arise because GPs become *convex optimization problems* via a logarithmic transformation. In addition to their mathematical benefits, recent research has shown that many practical problems can be formulated as GPs or closely approximated as GPs.

1.3 What are Signomials / Signomial Programs?

When the coefficients in a posynomial are allowed to be negative (but the variables stay strictly positive), that is called a Signomial. A Signomial Program has signomial constraints, and while they cannot be solved as quickly or to global optima, because they build on the structure of a GP they can be solved more quickly than a generic nonlinear program. More information on Signomial Programs can be found under **‘Advanced Commands‘_**.

1.4 Where can I learn more?

To learn more about GPs, refer to the following resources:

- A tutorial on geometric programming, by S. Boyd, S.J. Kim, L. Vandenberghe, and A. Hassibi.
- Convex optimization, by S. Boyd and L. Vandenberghe.
- Geometric Programming for Aircraft Design Optimization, Hoburg, Abbeel 2014

CHAPTER 2

GPkit Overview

GPkit is a Python package for defining and manipulating geometric programming (GP) models, abstracting away the backend solver.

The goal of GPkit is to make it easy to create, share, and explore geometric programming models, which tends to align well with being fast and mathematically correct.

2.1 Symbolic expressions

GPkit is a limited symbolic algebra language, allowing only for the creation of geometric program compatible equations (or signomial program compatible ones, if signomial programming is enabled). As mentioned in **'Geometric Programming 101'_**, one can view monomials as posynomials with a single term, and posynomials as signomials that have only positive coefficients. The inheritance structure of these objects in GPkit follows this mathematical basis.

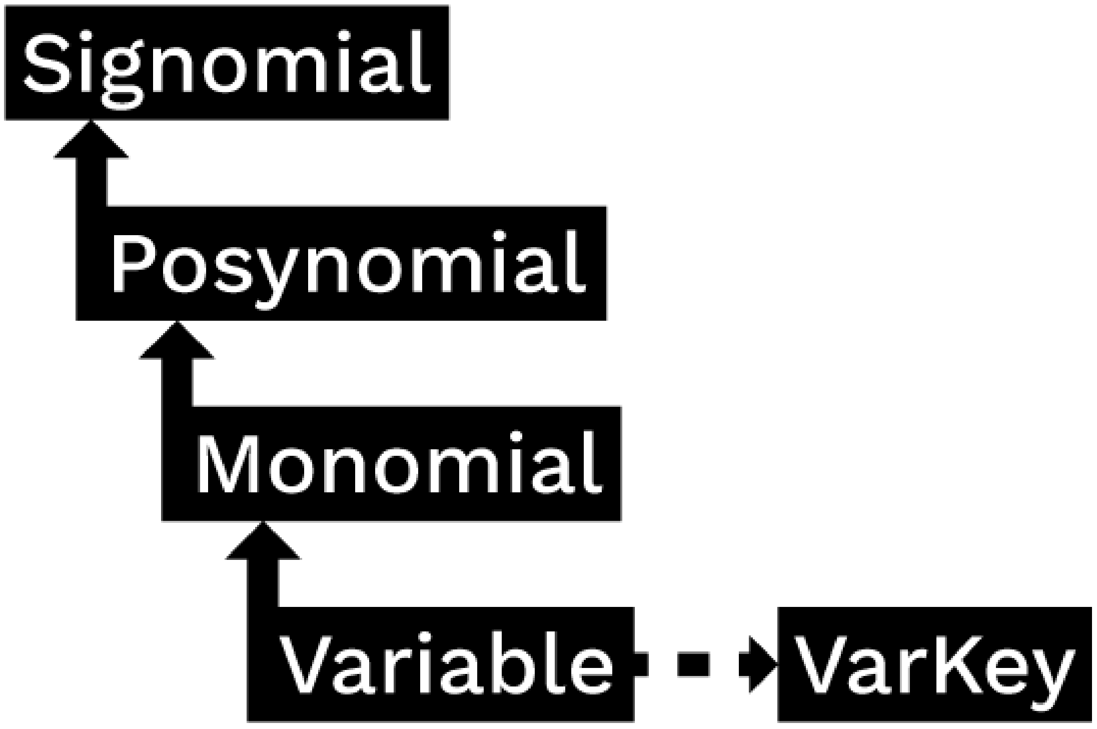
2.2 Substitution

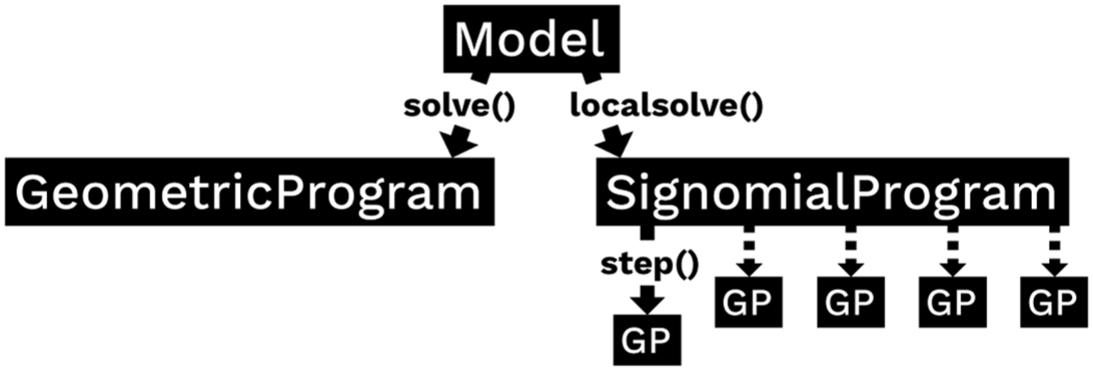
The `Varkey` object in the graph above is not a algebraic expression, but what GPkit uses as a variable's "name". It carries the LaTeX representation of a variable and its units, as well as any other information the user wishes to associate with a variable. The use of `VarKeys` as opposed to numeric indexing is an important part of the GPkit framework, because it allows a user to keep variable information local and modular.

GPkit keeps its internal representation of objects entirely symbolic until it solves. This means that any expression or Model object can replace any instance of a variable (as represented by a VarKey) with a number, new VarKey, or even an entire Monomial at any time with the `.sub()` method.

2.3 Model objects

In GPkit, a `Model` object represents a symbolic problem declaration. That problem may be either GP-compatible or SP-compatible. To avoid confusion, calling the `solve()` method on a model will either attempt to solve it for a global optimum (if it's a GP) or return an error immediately (if it's an SP). Similarly, calling `localsolve()` will either start the process of SP-solving (stepping through a sequence of GP-approximations) or return an error for GP-compatible Models. This framework is illustrated below.


Signomial
Posynomial
Monomial
Variable
VarKey


Model
solve()
localsolve()
GeometricProgram
SignomialProgram
step()
GP
GP
GP
GP
GP

CHAPTER 3

Installation Instructions

If you encounter bugs during installation email `gpkit@mit.edu`, or raise a new issue.

3.1 Install dependencies

GPkit's dependencies are the python packages

- `pip`
- `numpy` version 1.8 or newer
- `scipy`
- `pint`

and at least one solver (which we will install in a later step).

There are many ways to install these dependencies. Below is one suggestion for how to do so.

3.1.1 Get `pip`

Mac OS X Run `easy_install pip` at a terminal window.

Linux

Use your package manager to install `pip` Ubuntu: `sudo apt-get install python-pip`

Windows Do nothing at this step.

3.1.2 Get python packages

Mac OS X

Run the following commands:

- `pip install pip --upgrade`
- `pip install numpy`
- `pip install scipy`
- `pip install pint`

Linux

Use your package manager to install `numpy` and `scipy` Ubuntu: `sudo apt-get install python-numpy python-scipy`

Run `pip install pint`

Windows Do nothing at this step.

3.2 Install a GP solver

GPkit interfaces with two off the shelf solvers: cvxopt, and mosek. Cvxopt is open source; mosek requires a commercial licence or (free) academic license.

At least one solver is required.

Unfortunately, on Windows, due to 32-bit vs 64 bit issues, we do not currently know of a way to install both cvxopt and mosek simultaneously. If you are a Windows user, you should pick one solver or the other. For Windows 10, cvxopt does not appear to be an option.

3.2.1 Installing cvxopt

Mac OSX and Linux Run `pip install cvxopt`

Windows If you are using Windows 10, stop. Go to *Installing mosek*.

Install the Python 2.7 version of Python (x,y) (note that Python (x,y) is 32-bit)

- Installing CVXOPT with Anaconda or another Python distribution can be difficult, which is why we recommend Python (x,y).
- Python (x,y) recommends removing any previous installations of Python before installation.
- Be sure to click the cvxopt and pint check boxes under "Choose components" during installation.

3.2.2 Installing mosek

Note: if you do not have a paid license, you will need an academic or trial license to proceed.

Mac OS X

- If `which gcc` does not return anything, install `XCode` and the Apple Command Line Tools.
- Install cytypesgen via `pip install ctypesgen --pre` (gpkit uses ctypesgen to interface with the MOSEK C bindings).
- **Download MOSEK, then:**
  - Move the `mosek` folder to your home directory
  - Follow these steps for Mac.
  - Request an academic license file and put it in `~/mosek/`

Linux

- Install cytypesgen via `pip install ctypesgen --pre` (gpkit uses ctypesgen to interface with the MOSEK C bindings).
- **Download MOSEK, then:**

   - Move the `mosek` folder to your home directory
   - Follow these steps for Linux.
   - Request an academic license file and put it in `~/mosek/`

Windows If you have a 32-bit version of Windows, stop. Go to *Installing cvxopt*.

- Install the 64-bit version of Anaconda.
- Install cytypesgen via `pip install ctypesgen --pre` (gpkit uses ctypesgen to interface with the MOSEK C bindings).
- **Download MOSEK, then:**
   - Follow these steps for Windows.
   - Request an academic license file and put it in `C:\Users\(your_username)\mosek\`
   - **Make sure `gcc` is on your system path.**
      * To do this, type `gcc` into a command prompt.
      * If you get `executable not found`, then install the 64-bit version of mingw.
      * Make sure the `mingw` bin directory is on your system path (you may have to add it manually).

3.3 Install GPkit

- Run `pip install gpkit` at the command line.
- Run `pip install ipywidgets` for interactive control of models (recommended)
- Run `python -c "import gpkit.tests; gpkit.tests.run()"`
- *Optional:* to install gpkit into an isolated python environment, install virtualenv, run `virtualenv $DESTINATION_DIR` then activate it with `source $DESTINATION_DIR/bin/activate`

3.4 Debugging installation

You may need to rebuild GPkit if any of the following occur:

- You install a new solver (mosek or cvxopt) after installing GPkit
- You delete the `.gpkit` folder from your home directory
- You see `Could not load settings file.`
- You see `Could not load MOSEK library: ImportError('$HOME/.gpkit/expopt.so not found.')`

To rebuild GPkit, do the following:

- Run `pip uninstall gpkit`
- Run `pip install --no-cache-dir --no-deps gpkit`
- Run `python -c "import gpkit.tests; gpkit.tests.run()"`
- If any tests fail, email `gpkit@mit.edu`

3.5 Updating GPkit between releases

Active developers may wish to install the latest GPkit directly from the source code on Github. To do so,

1. Run `pip uninstall gpkit` to uninstall your existing GPkit.
2. Run `git clone https://github.com/hoburg/gpkit.git` to clone the GPkit repository, or `cd gpkit; git pull origin master; cd ..` to update your existing repository.
3. Run `pip install -e gpkit` to reinstall GPkit.
4. Run `python -c "import gpkit.tests; gpkit.tests.run()"` to test your installation.

CHAPTER 4

Getting Started

GPkit is a Python package, so we assume basic familiarity with Python: if you're new to Python we recommend you take a look at Learn Python.

Alright: install GPkit and import away.

```python
from gpkit import Variable, VectorVariable, Model
```

4.1 Declaring Variables

Instances of the `Variable` class represent scalar variables. They store a key (i.e. name) used to look up the Variable in dictionaries, and optionally units, a description, and a value (if the Variable is to be held constant).

4.1.1 Free Variables

```python
# Declare a variable, x
x = Variable("x")

# Declare a variable, y, with units of meters
y = Variable("y", "m")

# Declare a variable, z, with units of meters, and a description
z = Variable("z", "m", "A variable called z with units of meters")
```

4.1.2 Fixed Variables

To declare a variable with a constant value, use the `Variable` class, as above, but specify the `value=` input argument:

```python
# Declare \rho equal to 1.225 kg/m^3.
# NOTE: write a literal backslash by preceding it with another backslash
rho = Variable("\\rho", 1.225, "kg/m^3", "Density of air at sea level")
```

In the example above, the key name `"\\rho"` is for LaTeX printing (described later). The unit and description arguments are optional.

```python
#Declare pi equal to 3.14
pi = Variable("\\pi", 3.14)
```

4.1.3 Vector Variables

Vector variables are represented by the `VectorVariable` class. The first argument is the length of the vector. All other inputs follow those of the `Variable` class.

```
# Declare a 3-element vector variable "x" with units of "m"
x = VectorVariable(3, "x", "m", "Cube corner coordinates")
x_min = VectorVariable(3, "x", [1, 2, 3], "m", "Cube corner minimum")
```

4.2 Creating Monomials and Posynomials

Monomial and posynomial expressions can be created using mathematical operations on variables.

```
# create a Monomial term xy^2/z
x = Variable("x")
y = Variable("y")
z = Variable("z")
m = x * y**2 / z
type(m)  # gpkit.nomials.Monomial
```

```
# create a Posynomial expression x + xy^2
x = Variable("x")
y = Variable("y")
p = x + x * y**2
type(p)  # gpkit.nomials.Posynomial
```

4.3 Declaring Constraints

`Constraint` objects represent constraints of the form `Monomial >= Posynomial` or `Monomial == Monomial` (which are the forms required for GP-compatibility).

Note that constraints must be formed using <=, >=, or == operators, not < or >.

```
# consider a block with dimensions x, y, z less than 1
# constrain surface area less than 1.0 m^2
x = Variable("x", "m")
y = Variable("y", "m")
z = Variable("z", "m")
S = Variable("S", 1.0, "m^2")
c = (2*x*y + 2*x*z + 2*y*z <= S)
type(c)  # gpkit.nomials.PosynomialInequality
```

4.4 Formulating a Model

The `Model` class represents an optimization problem. To create one, pass an objective and list of Constraints.

By convention, the objective is the function to be *minimized*. If you wish to *maximize* a function, take its reciprocal. For example, the code below creates an objective which, when minimized, will maximize `x*y*z`.

```
objective = 1/(x*y*z)
constraints = [2*x*y + 2*x*z + 2*y*z <= S,
               x >= 2*y]
m = Model(objective, constraints)
```

4.5 Solving the Model

When solving the model you can change the level of information that gets printed to the screen with the `verbosity` setting. A verbosity of 1 (the default) prints warnings and the solution; a verbosity of 2 prints solve time, a verbosity of 3 prints solver output, and a verbosity of 0 prints nothing.

```
sol = m.solve(verbosity=0)
```

4.6 Printing Results

We can also manually print the solution table, with the same result as if the verbosity argument had been left blank above.

```
print sol.table()
```

```
Cost
----
 15.59 [1/m**3]

Free Variables
--------------
x : 0.5774  [m]
y : 0.2887  [m]
z : 0.3849  [m]

Constants
---------
S : 1  [m**2]

Sensitivities
-------------
S : -1.5
```

```
print "The x dimension is %s." % (sol(x))
```

```
The x dimension is 0.577351209028 meter.
```

4.7 Sensitivities and dual variables

When a GP is solved, the solver returns not just the optimal value for the problem's variables (known as the "primal solution") but also, as a side effect of the solving process, the effect that scaling the ≤ 1 of each canonical constraint would have on the overall objective (the "dual solution").

From the dual solution GPkit computes the sensitivities for every fixed variable in the problem. This can be quite useful for seeing which constraints are most crucial, and prioritizing remodeling and assumption-checking.

4.7.1 Using variable sensitivities

Fixed variable sensitivities can be accessed most easily using a SolutionArray's `sens()` method, as in this example:

```
import gpkit
x = gpkit.Variable("x")
x_min = gpkit.Variable("x_{min}", 2)
sol = gpkit.Model(x, [x_min <= x]).solve()
assert sol.sens(x_min) == 1
```

These sensitivities are actually log derivatives ($\frac{d\log(y)}{d\log(x)}$); whereas a regular derivative is a tangent line, these are tangent monomials, so the `1` above indicates that `x_min` has a linear relation with the objective. This is confirmed by a further example:

```
import gpkit
x = gpkit.Variable("x")
x_squared_min = gpkit.Variable("x^2_{min}", 2)
sol = gpkit.Model(x, [x_squared_min <= x**2]).solve()
assert sol.sens(x_squared_min) == 2
```

CHAPTER 5

Advanced Commands

5.1 Feasibility Analysis

If your Model doesn't solve, you can automatically find the nearest feasible version of it with the `Model.feasibility()` command, as shown below. The feasible version can either involve relaxing all constraints by the smallest number possible (that is, dividing the less-than side of every constraint by the same number), relaxing each constraint by its own number and minimizing the product of those numbers, or changing each constant by the smallest total percentage possible.

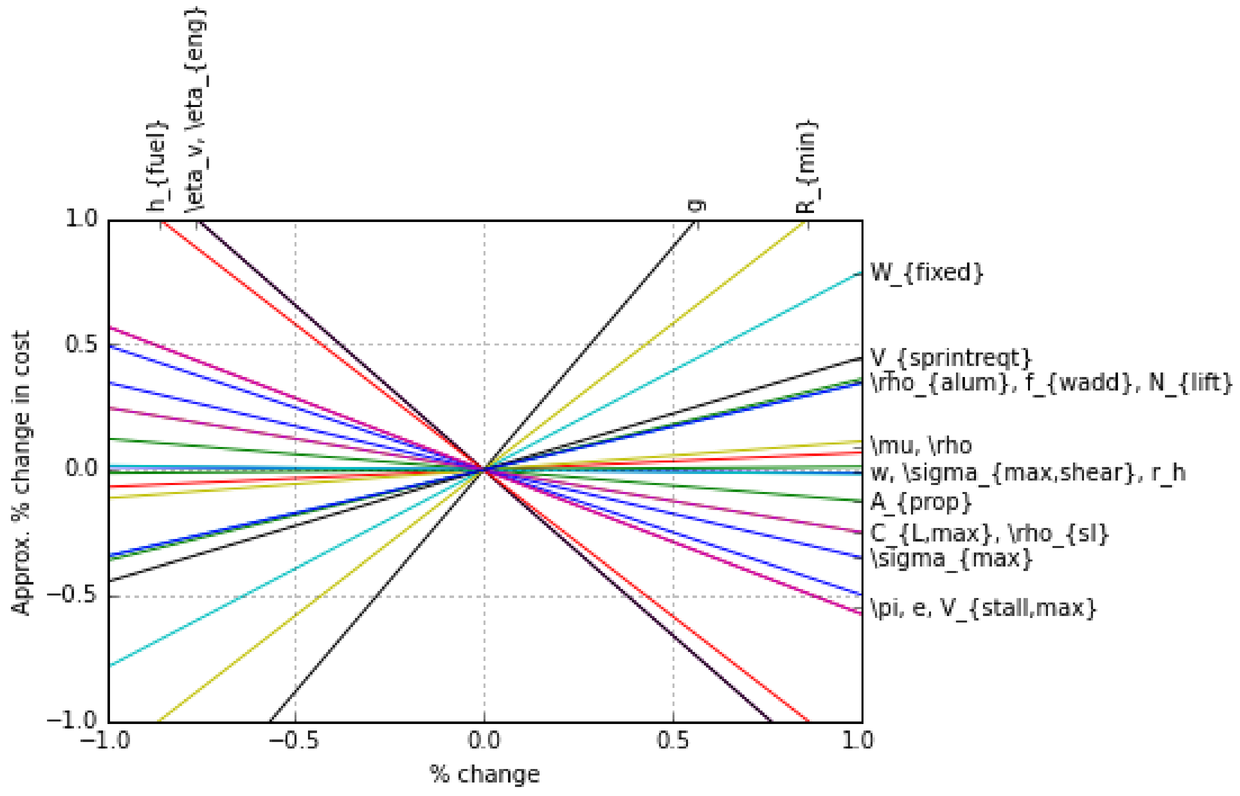
```
from gpkit import Variable, Model, NomialArray
x = Variable("x")
x_min = Variable("x_min", 2)
x_max = Variable("x_max", 1)
m = Model(x, [x <= x_max, x >= x_min])
# m.solve()  # raises a RuntimeWarning!
feas = m.feasibility()

# USING OVERALL
m.constraints = NomialArray(m.signomials)/feas["overall"]
m.solve()

# USING CONSTRAINTS
m = Model(x, [x <= x_max, x >= x_min])
m.constraints = NomialArray(m.signomials)/feas["constraints"]
m.solve()

# USING CONSTANTS
m = Model(x, [x <= x_max, x >= x_min])
m.substitutions.update(feas["constants"])
m.solve()
```

5.2 Plotting variable sensitivities

Sensitivities are a useful way to evaluate the tradeoffs in your model, as well as what aspects of the model are driving the solution and should be examined. To help with this, GPkit has an automatic sensitivity plotting function that can be accessed as follows:

```
from gpkit.interactive.plotting import sensitivity_plot
sensitivity_plot(m)
```

Which produces the following plot:



In this plot, steep lines that go up to the right are variables whose increase sharply increases (makes worse) the objective. Steep lines going down to the right are variables whose increase sharply decreases (improves) the objective.

5.3 Substitutions

Substitutions are a general-purpose way to change every instance of one variable into either a number or another variable.

5.3.1 Substituting into Posynomials, NomialArrays, and GPs

The examples below all use Posynomials and NomialArrays, but the syntax is identical for GPs (except when it comes to sweep variables).

```
# adapted from t_sub.py / t_NomialSubs / test_Basic
from gpkit import Variable
x = Variable("x")
p = x**2
assert p.sub(x, 3) == 9
assert p.sub(x.varkeys["x"], 3) == 9
assert p.sub("x", 3) == 9
```

Here the variable `x` is being replaced with `3` in three ways: first by substituting for `x` directly, then by substituting for the `VarKey("x")`, then by substituting the string “x”. In all cases the substitution is understood as being with the VarKey: when a variable is passed in the VarKey is pulled out of it, and when a string is passed in it is used as an argument to the Posynomial's `varkeys` dictionary.

5.3.2 Substituting multiple values

```
# adapted from t_sub.py / t_NomialSubs / test_Vector
from gpkit import Variable, VectorVariable
x = Variable("x")
y = Variable("y")
z = VectorVariable(2, "z")
p = x*y*z
assert all(p.sub({x: 1, "y": 2}) == 2*z)
assert all(p.sub({x: 1, y: 2, "z": [1, 2]}) == z.sub(z, [2, 4]))
```

To substitute in multiple variables, pass them in as a dictionary where the keys are what will be replaced and values are what it will be replaced with. Note that you can also substitute for VectorVariables by their name or by their NomialArray.

5.3.3 Substituting with nonnumeric values

You can also substitute in sweep variables (see *Sweeps*), strings, and monomials:

```
# adapted from t_sub.py / t_NomialSubs
from gpkit import Variable
from gpkit.small_scripts import mag

x = Variable("x", "m")
xvk = x.varkeys.values()[0]
descr_before = x.exp.keys()[0].descr
y = Variable("y", "km")
yvk = y.varkeys.values()[0]
for x_ in ["x", xvk, x]:
    for y_ in ["y", yvk, y]:
        if not isinstance(y_, str) and type(xvk.units) != str:
            expected = 0.001
        else:
            expected = 1.0
        assert abs(expected - mag(x.sub(x_, y_).c)) < 1e-6
if type(xvk.units) != str:
    # this means units are enabled
    z = Variable("z", "s")
    # y.sub(y, z) will raise ValueError due to unit mismatch
```

Note that units are preserved, and that the value can be either a string (in which case it just renames the variable), a varkey (in which case it changes its description, including the name) or a Monomial (in which case it substitutes for the variable with a new monomial).

5.3.4 Substituting with replacement

Any of the substitutions above can be run with `p.sub(*args, replace=True)` to clobber any previously-substitued values.

5.3.5 Fixed Variables

When a Model is created, any fixed Variables are used to form a dictionary: `{var: var.descr["value"] for var in self.varlocs if "value" in var.descr}`. This dictionary in then substituted into the Model's cost and constraints before the `substitutions` argument is (and hence values are supplanted by any later substitutions).

`solution.subinto(p)` will substitute the solution(s) for variables into the posynomial `p`, returning a NomialArray. For a non-swept solution, this is equivalent to `p.sub(solution["variables"])`.

You can also substitute by just calling the solution, i.e. `solution(p)`. This returns a numpy array of just the coefficients (`c`) of the posynomial after substitution, and will raise a‘ `ValueError`‘ if some of the variables in `p` were not found in `solution`.

5.3.6 Freeing Fixed Variables

After creating a Model, it may be useful to "free" a fixed variable and resolve. This can be done using the command `del m.substitutions["x"]`, where `m` is a Model. An example of how to do this is shown below.

```
from gpkit import Variable, Model
x = Variable("x")
y = Variable("y", 3)  # fix value to 3
m = Model(x, [x >= 1 + y, y >= 1])
_ = m.solve()  # optimal cost is 4; y appears in Constants

del m.substitutions["y"]
_ = m.solve()  # optimal cost is 2; y appears in Free Variables
```

Note that `del m.substitutions["y"]` affects `m` but not `y.key`. `y.value` will still be 3, and if `y` is used in a new model, it will still carry the value of 3.

5.4 Sweeps

5.4.1 Declaring Sweeps

Sweeps are useful for analyzing tradeoff surfaces. A sweep "value" is an Iterable of numbers, e.g. `[1, 2, 3]`. Variables are swept when their substitution value takes the form `('sweep', Iterable), (e.g. 'sweep', np.linspace(1e6, 1e7, 100))`. During variable declaration, giving an Iterable value for a Variable is assumed to be giving it a sweeep value: for example, `x = Variable("x", [1, 2, 3]`. Sweeps can also be declared during later substitution (`gp.sub("x", ('sweep', [1, 2, 3]))`, or if the variable was already substituted for a constant, `gp.sub("x", ('sweep', [1, 2, 3]), replace=True))`.

5.4.2 Solving Sweeps

A Model with sweeps will solve for all possible combinations: e.g., if there's a variable `x` with value `('sweep', [1, 3])` and a variable `y` with value `('sweep', [14, 17])` then the gp will be solved four times, for $(x, y) \in \{(1, 14),\ (1, 17),\ (3, 14),\ (3, 17)\}$. The returned solutions will be a one-dimensional array (or 2-D for vector variables), accessed in the usual way. Sweeping Vector Variables

Vector variables may also be substituted for: `y = VectorVariable(3, "y", value=('sweep' ,[[1, 2], [1, 2], [1, 2]])` will sweep $y\ \forall\ y_i \in \{1, 2\}$.

5.4.3 Parallel Sweeps

During a normal sweep, each result is independent, so they can be run in parallel. To use this feature, run `$ ipcluster start` at a terminal: it will automatically start a number of iPython parallel computing engines equal to the number of cores on your machine, and when you next import gpkit you should see a note like `Using parallel execution of sweeps on 4 clients`. If you do, then all sweeps performed with that import of gpkit will be parallelized.

This parallelization sets the stage for gpkit solves to be outsourced to a server, which may be valuable for faster results; alternately, it could allow the use of gpkit without installing a solver.

5.4.4 Linked Sweeps

Some constants may be "linked" to another sweep variable. This can be represented by a Variable whose value is `('sweep', fn)`, where the arguments of the function `fn` are stored in the Varkeys's `args` attribute. If you declare a variables value to be a function, then it will assume you meant that as a sweep value: for example, `a_ = gpkit.Variable("a_", lambda a: 1-a, "-", args=[a])` will create a constant whose value is always 1 minus the value of a (valid for values of a less than 1). Note that this declaration requires the variable `a` to already have been declared.

5.4.5 Example Usage

```
# code from t_GPSubs.test_VectorSweep in tests/t_sub.py
from gpkit import Variable, VectorVariable, Model

x = Variable("x")
y = VectorVariable(2, "y")
m = Model(x, [x >= y.prod()])
m.substitutions.update({y: ('sweep', [[2, 3], [5, 7, 11]])})
a = m.solve(printing=False)["cost"]
b = [10, 14, 22, 15, 21, 33]
assert all(abs(a-b)/(a+b) < 1e-7)
```

5.5 Composite Objectives

Given n posynomial objectives g_i, you can sweep out the problem's Pareto frontier with the composite objective:

$$g_0 w_0 \prod_{i \neq 0} v_i + g_1 w_1 \prod_{i \neq 1} v_i + ... + g_n \prod_i v_i$$

where $i \in 0...n-1$ and $v_i = 1 - w_i$ and $w_i \in [0,1]$

GPkit has the helper function `composite_objective` for constructing these.

5.5.1 Example Usage

```
import numpy as np
import gpkit

L, W = gpkit.Variable("L"), gpkit.Variable("W")

eqns = [L >= 1, W >= 1, L*W == 10]
```

```
co_sweep = [0] + np.logspace(-6, 0, 10).tolist()

obj = gpkit.tools.composite_objective(L+W, W**-1 * L**-3,
                                      normsub={L:10, W: 10},
                                      sweep=co_sweep)

m = gpkit.Model(obj, eqns)
m.solve()
```

The `normsub` argument specifies an expected value for your solution to normalize the different g_i (you can also do this by hand). The feasibility of the problem should not depend on the normalization, but the spacing of the sweep will.

The `sweep` argument specifies what points between 0 and 1 you wish to sample the weights at. If you want different resolutions or spacings for different weights, the `sweeps` argument accepts a list of sweep arrays.

CHAPTER 6

Signomial Programming

Signomial programming finds a local solution to a problem of the form:

$$\begin{array}{lll} \text{minimize} & g_0(x) & \\ \text{subject to} & f_i(x) = 1, & i = 1, ..., m \\ & g_i(x) - h_i(x) \leq 1, & i = 1, ..., n \end{array}$$

where each f is monomial while each g and h is a posynomial.

This requires multiple solutions of geometric programs, and so will take longer to solve than an equivalent geometric programming formulation.

The specification of a signomial problem can affect its solve time in a nuanced way: `gpkit.SP(x, [x >= 0.1, x+y >= 1, y <= 0.1]).localsolve()` takes about twice as long to solve with cvxopt as `gpkit.SP(x, [x >= 1-y, y <= 0.1]).localsolve()`, despite the two formulations being arithmetically equivalent and taking the same number of iterations.

In general, when given the choice of which variables to include in the positive-posynomial / g side of the constraint, the modeler should:

1. maximize the number of variables in g,
2. prioritize variables that are in the objective,
3. then prioritize variables that are present in other constraints.

The syntax `SP.localsolve` is chosen to emphasize that signomial programming returns a local optimum. For the same reason, calling `SP.solve` will raise an error.

By default, signomial programs are first solved conservatively (by assuming each h is equal only to its constant portion) and then become less conservative on each iteration.

6.1 Example Usage

```python
"""Adapted from t_SP in tests/t_geometric_program.py"""
import gpkit

# Decision variables
x = gpkit.Variable('x')
y = gpkit.Variable('y')

# must enable signomials for subtraction
with gpkit.SignomialsEnabled():
    constraints = [x >= 1-y, y <= 0.1]
```

```
# create and solve the SP
m = gpkit.Model(x, constraints)
sol = m.localsolve(verbosity=1)
assert abs(sol(x) - 0.9) < 1e-6
```

When using the `localsolve` method, the `reltol` argument specifies the relative tolerance of the solver: that is, by what percent does the solution have to improve between iterations? If any iteration improves less than that amount, the solver stops and returns its value.

If you wish to start the local optimization at a particular point x_k, however, you may do so by putting that position (a dictionary formatted as you would a substitution) as the `xk` argument.

6.2 Calling to External Codes

For some applications, it is useful to call external codes which may not be GP compatible. Imagine we wished to solve the following optimization problem:

$$\begin{array}{ll} \text{minimize} & y \\ \text{subject to} & y \geq \sin(x) \\ & \frac{\pi}{4} \leq x \leq \frac{\pi}{2} \end{array}$$

This problem is not GP compatible due to the sin(x) constraint. One approach might be to take the first term of the Taylor expansion of sin(x) and attempt to solve:

```
"Can be found in gpkit/docs/source/examples/sin_approx_example.py"
import numpy as np
from gpkit import Variable, Model


x = Variable("x")
y = Variable("y")

objective = y

constraints = [y >= x,
               x <= np.pi/2.,
               x >= np.pi/4.,
              ]

m = Model(objective, constraints)
sol = m.solve(verbosity=1)
```

```
Cost
----
 0.7854

Free Variables
--------------
x : 0.7854
y : 0.7854
```

We can do better, however, by utilizing some built in functionality of GPkit. Assume we have some external code which is capable of evaluating our incompatible function:

```
"""External function for GPkit to call.  Can be found
in gpkit/docs/source/examples/external_function.py"""
import numpy as np

def external_code(x):
    "Returns sin(x)"
    return np.sin(x)
```

Now, we can create a ConstraintSet that allows GPkit to treat the incompatible constraint as though it were a signomial programming constraint:

```
"Can be found in gpkit/docs/source/examples/external_constraint.py"
from gpkit import ConstraintSet
from external_function import external_code


class ExternalConstraint(ConstraintSet):
    "Class for external calling"
    # Overloading the __init__ function here permits the constraint class to be
    # called more cleanly at the top level GP.
    def __init__(self, x, y, **kwargs):

        # Calls the ConstriantSet __init__ function
        super(ExternalConstraint, self).__init__([], **kwargs)

        # We need a GPkit variable defined to return in our constraint.  The
        # easiest way to do this is to read in the parameters of interest in
        # the initiation of the class and store them here.
        self.x = x
        self.y = y

    # Prevents the ExternalConstraint class from solving in a GP, thus forcing
    # iteration
    def as_posyslt1(self):
        raise TypeError("ExternalConstraint Model cannot solve as a GP.")

    # Returns the ExternalConstraint class as a GP compatible constraint when
    # requested by the GPkit solver
    def as_gpconstr(self, x0):

        # Unpacking the GPkit variables
        x = self.x
        y = self.y

        # Creating a default constraint for the first solve
        if not x0:
            return (y >= x)

        # Returns constraint updated with new call to the external code
        else:
            # Unpack Design Variables at the current point
            x_star = x0["x"]

            # Call external code
            res = external_code(x_star)

        # Return linearized constraint
        return (y >= res*x/x_star)
```

and replace the incompatible constraint in our GP:

```
"Can be found in gpkit/docs/source/examples/external_sp.py"

import numpy as np
from gpkit import Variable, Model
from external_constraint import ExternalConstraint

x = Variable("x")
y = Variable("y")

objective = y

constraints = [ExternalConstraint(x, y),
               x <= np.pi/2.,
               x >= np.pi/4.,
              ]

m = Model(objective, constraints)
sol = m.localsolve(verbosity=1)
```

```
Cost
----
 0.7071

Free Variables
--------------
x : 0.7854
y : 0.7071
```

which is the expected result. This method has been generalized to larger problems, such as calling XFOIL and AVL.

If you wish to start the local optimization at a particular point x_0, however, you may do so by putting that position (a dictionary formatted as you would a substitution) as the `x0` argument

CHAPTER 7

Examples

7.1 iPython Notebook Examples

More examples, including some with in-depth explanations and interactive visualizations, can be seen on nbviewer.

7.2 A Trivial GP

The most trivial GP we can think of: minimize x subject to the constraint $x \geq 1$.

```
"Very simple problem: minimize x while keeping x greater than 1."
from gpkit import Variable, Model

# Decision variable
x = Variable('x')

# Constraint
constraints = [x >= 1]

# Objective (to minimize)
objective = x

# Formulate the Model
m = Model(objective, constraints)

# Solve the Model
sol = m.solve(verbosity=0)

# print selected results
print("Optimal cost:  %s" % sol['cost'])
print("Optimal x val: %s" % sol(x))
```

Of course, the optimal value is 1. Output:

```
Optimal cost:  1.0
Optimal x val: 1.0
```

7.3 Maximizing the Volume of a Box

This example comes from Section 2.4 of the GP tutorial, by S. Boyd et. al.

```
"Maximizes box volume given area and aspect ratio constraints."
from gpkit import Variable, Model

# Parameters
alpha = Variable("alpha", 2, "-", "lower limit, wall aspect ratio")
beta = Variable("beta", 10, "-", "upper limit, wall aspect ratio")
gamma = Variable("gamma", 2, "-", "lower limit, floor aspect ratio")
delta = Variable("delta", 10, "-", "upper limit, floor aspect ratio")
A_wall = Variable("A_{wall}", 200, "m^2", "upper limit, wall area")
A_floor = Variable("A_{floor}", 50, "m^2", "upper limit, floor area")

# Decision variables
h = Variable("h", "m", "height")
w = Variable("w", "m", "width")
d = Variable("d", "m", "depth")

#Constraints
constraints = [A_wall >= 2*h*w + 2*h*d,
               A_floor >= w*d,
               h/w >= alpha,
               h/w <= beta,
               d/w >= gamma,
               d/w <= delta]

#Objective function
V = h*w*d
objective = 1/V  # To maximize V, we minimize its reciprocal

# Formulate the Model
m = Model(objective, constraints)

# Solve the Model and print the results table
sol = m.solve(verbosity=1)
```

The output is

```
Cost
----
 0.003674 [1/m**3]

Free Variables
--------------
d : 8.17   [m] depth
h : 8.163  [m] height
w : 4.081  [m] width

Constants
---------
A_{floor} : 50   [m**2] upper limit, floor area
 A_{wall} : 200  [m**2] upper limit, wall area
    alpha : 2           lower limit, wall aspect ratio
     beta : 10          upper limit, wall aspect ratio
    delta : 10          upper limit, floor aspect ratio
    gamma : 2           lower limit, floor aspect ratio
```

```
Sensitivities
-------------
   alpha : 0.5  lower limit, wall aspect ratio
A_{wall} : -1.5 upper limit, wall area
```

7.4 Water Tank

Say we had a fixed mass of water we wanted to contain within a tank, but also wanted to minimize the cost of the material we had to purchase (i.e. the surface area of the tank):

```
"Minimizes cylindrical tank surface area for a particular volume."
from gpkit import Variable, VectorVariable, Model

M = Variable("M", 100, "kg", "Mass of Water in the Tank")
rho = Variable("\\rho", 1000, "kg/m^3", "Density of Water in the Tank")
A = Variable("A", "m^2", "Surface Area of the Tank")
V = Variable("V", "m^3", "Volume of the Tank")
d = VectorVariable(3, "d", "m", "Dimension Vector")

constraints = (A >= 2*(d[0]*d[1] + d[0]*d[2] + d[1]*d[2]),
               V == d[0]*d[1]*d[2],
               M == V*rho)

m = Model(A, constraints)
sol = m.solve(verbosity=1)
```

The output is

```
Cost
----
 1.293 [m**2]

Free Variables
--------------
      A : 1.293                                [m**2] Surface Area of the Tank
      V : 0.1                                  [m**3] Volume of the Tank
\vec{d} : [ 0.464     0.464     0.464     ]   [m]    Dimension Vector

Constants
---------
   M : 100   [kg]      Mass of Water in the Tank
\rho : 1000  [kg/m**3] Density of Water in the Tank

Sensitivities
-------------
   M : 0.6667  Mass of Water in the Tank
\rho : -0.6667 Density of Water in the Tank
```

7.5 Simple Wing

This example comes from Section 3 of Geometric Programming for Aircraft Design Optimization, by W. Hoburg and P. Abbeel.

```
"Minimizes airplane drag for a simple drag and structure model."
import numpy as np
from gpkit import Variable, Model


# Constants
k = Variable("k", 1.2, "-", "form factor")
e = Variable("e", 0.95, "-", "Oswald efficiency factor")
mu = Variable("\\mu", 1.78e-5, "kg/m/s", "viscosity of air")
pi = Variable("\\pi", np.pi, "-", "half of the circle constant")
rho = Variable("\\rho", 1.23, "kg/m^3", "density of air")
tau = Variable("\\tau", 0.12, "-", "airfoil thickness to chord ratio")
N_ult = Variable("N_{ult}", 3.8, "-", "ultimate load factor")
V_min = Variable("V_{min}", 22, "m/s", "takeoff speed")
C_Lmax = Variable("C_{L,max}", 1.5, "-", "max CL with flaps down")
S_wetratio = Variable("(\\frac{S}{S_{wet}})", 2.05, "-", "wetted area ratio")
W_W_coeff1 = Variable("W_{W_{coeff1}}", 8.71e-5, "1/m",
                      "Wing Weight Coefficent 1")
W_W_coeff2 = Variable("W_{W_{coeff2}}", 45.24, "Pa",
                      "Wing Weight Coefficent 2")
CDA0 = Variable("(CDA0)", 0.031, "m^2", "fuselage drag area")
W_0 = Variable("W_0", 4940.0, "N", "aircraft weight excluding wing")

# Free Variables
D = Variable("D", "N", "total drag force")
A = Variable("A", "-", "aspect ratio")
S = Variable("S", "m^2", "total wing area")
V = Variable("V", "m/s", "cruising speed")
W = Variable("W", "N", "total aircraft weight")
Re = Variable("Re", "-", "Reynold's number")
C_D = Variable("C_D", "-", "Drag coefficient of wing")
C_L = Variable("C_L", "-", "Lift coefficent of wing")
C_f = Variable("C_f", "-", "skin friction coefficient")
W_w = Variable("W_w", "N", "wing weight")

constraints = []

# Drag model
C_D_fuse = CDA0/S
C_D_wpar = k*C_f*S_wetratio
C_D_ind = C_L**2/(pi*A*e)
constraints += [C_D >= C_D_fuse + C_D_wpar + C_D_ind]

# Wing weight model
W_w_strc = W_W_coeff1*(N_ult*A**1.5*(W_0*W*S)**0.5)/tau
W_w_surf = W_W_coeff2 * S
constraints += [W_w >= W_w_surf + W_w_strc]

# and the rest of the models
constraints += [D >= 0.5*rho*S*C_D*V**2,
                Re <= (rho/mu)*V*(S/A)**0.5,
                C_f >= 0.074/Re**0.2,
                W <= 0.5*rho*S*C_L*V**2,
                    W <= 0.5*rho*S*C_Lmax*V_min**2,
                    W >= W_0 + W_w]

print("SINGLE\n======")
m = Model(D, constraints)
sol = m.solve(verbosity=1)

print("SWEEP\n=====")
N = 2
sweeps = {V_min: ("sweep", np.linspace(20, 25, N)),
          V: ("sweep", np.linspace(45, 55, N)), }
m.substitutions.update(sweeps)
m.solve(verbosity=1)
```

The output is

```
SINGLE
======

Cost
----
 303.1 [N]

Free Variables
--------------
  A : 8.46                aspect ratio
C_D : 0.02059             Drag coefficient of wing
C_L : 0.4988              Lift coefficent of wing
C_f : 0.003599            skin friction coefficient
  D : 303.1      [N]      total drag force
 Re : 3.675e+06           Reynold's number
  S : 16.44      [m**2]   total wing area
  V : 38.15      [m/s]    cruising speed
  W : 7341       [N]      total aircraft weight
W_w : 2401       [N]      wing weight

Constants
---------
             (CDA0) : 0.031     [m**2]    fuselage drag area
(\frac{S}{S_{wet}}) : 2.05                wetted area ratio
          C_{L,max} : 1.5                 max CL with flaps down
            N_{ult} : 3.8                 ultimate load factor
            V_{min} : 22        [m/s]     takeoff speed
                W_0 : 4940      [N]       aircraft weight excluding wing
     W_{W_{coeff1}} : 8.71e-05  [1/m]     Wing Weight Coefficent 1
     W_{W_{coeff2}} : 45.24     [Pa]      Wing Weight Coefficent 2
                \mu : 1.78e-05  [kg/m/s]  viscosity of air
                \pi : 3.142               half of the circle constant
               \rho : 1.23      [kg/m**3] density of air
               \tau : 0.12                airfoil thickness to chord ratio
                  e : 0.95                Oswald efficiency factor
                  k : 1.2                 form factor

Sensitivities
-------------
                W_0 : 1.011   aircraft weight excluding wing
                  k : 0.4299  form factor
(\frac{S}{S_{wet}}) : 0.4299  wetted area ratio
     W_{W_{coeff1}} : 0.2903  Wing Weight Coefficent 1
```

```
             N_{ult} : 0.2903  ultimate load factor
      W_{W_{coeff2}} : 0.1303  Wing Weight Coefficent 2
              (CDA0) : 0.09156 fuselage drag area
                 \mu : 0.08599 viscosity of air
           C_{L,max} : -0.1839 max CL with flaps down
                \rho : -0.2269 density of air
                \tau : -0.2903 airfoil thickness to chord ratio
             V_{min} : -0.3678 takeoff speed
                   e : -0.4785 Oswald efficiency factor
                 \pi : -0.4785 half of the circle constant

SWEEP
=====

Cost
----
 [ 338       294       396       326       ] [N]

Sweep Variables
---------------
      V : [ 45        45        55        55       ]  [m/s] cruising speed
V_{min} : [ 20        25        20        25       ]  [m/s] takeoff speed

Free Variables
--------------
  A : [ 6.2       8.84      4.77      7.16     ]         aspect ratio
C_D : [ 0.0146    0.0196    0.0123    0.0157   ]         Drag coefficient of wing
C_L : [ 0.296     0.463     0.198     0.31     ]         Lift coefficent of wing
C_f : [ 0.00333   0.00361   0.00314   0.00342  ]         skin friction coefficient
  D : [ 338       294       396       326      ]  [N]    total drag force
 Re : [ 5.38e+06  3.63e+06  7.24e+06  4.75e+06 ]         Reynold's number
  S : [ 18.6      12.1      17.3      11.2     ]  [m**2] total wing area
  W : [ 6.85e+03  6.97e+03  6.4e+03   6.44e+03 ]  [N]    total aircraft weight
W_w : [ 1.91e+03  2.03e+03  1.46e+03  1.5e+03  ]  [N]    wing weight

Constants
---------
             (CDA0) : 0.031     [m**2]    fuselage drag area
(\frac{S}{S_{wet}}) : 2.05                wetted area ratio
          C_{L,max} : 1.5                 max CL with flaps down
            N_{ult} : 3.8                 ultimate load factor
                W_0 : 4940      [N]       aircraft weight excluding wing
     W_{W_{coeff1}} : 8.71e-05  [1/m]     Wing Weight Coefficent 1
     W_{W_{coeff2}} : 45.24     [Pa]      Wing Weight Coefficent 2
                \mu : 1.78e-05  [kg/m/s]  viscosity of air
                \pi : 3.142               half of the circle constant
               \rho : 1.23      [kg/m**3] density of air
               \tau : 0.12                airfoil thickness to chord ratio
                  e : 0.95                Oswald efficiency factor
                  k : 1.2                 form factor

Sensitivities
-------------
                W_0 : [ 0.919     0.947     0.845     0.847    ] aircraft weight excluding wing
                  V : [ 0.589     0.249     0.975     0.746    ] cruising speed
                  k : [ 0.561     0.454     0.63      0.536    ] form factor
(\frac{S}{S_{wet}}) : [ 0.561     0.454     0.63      0.536    ] wetted area ratio
     W_{W_{coeff1}} : [ 0.179     0.247     0.108     0.155    ] Wing Weight Coefficent 1
```

```
          N_{ult} : [ 0.179     0.247     0.108     0.155    ] ultimate load factor
           (CDA0) : [ 0.114     0.131     0.146     0.177    ] fuselage drag area
   W_{W_{coeff2}} : [ 0.141     0.0911    0.126     0.0787   ] Wing Weight Coefficent 2
              \mu : [ 0.112     0.0907    0.126     0.107    ] viscosity of air
             \rho : [ -0.172    -0.129    -0.097    -0.0331  ] density of air
             \tau : [ -0.179    -0.247    -0.108    -0.155   ] airfoil thickness to chord rati
                e : [ -0.325    -0.415    -0.225    -0.287   ] Oswald efficiency factor
              \pi : [ -0.325    -0.415    -0.225    -0.287   ] half of the circle constant
        C_{L,max} : [ -0.411    -0.207    -0.521    -0.353   ] max CL with flaps down
          V_{min} : [ -0.822    -0.415    -1.04     -0.705   ] takeoff speed
```

7.6 Simple Beam

In this example we consider a beam subjected to a uniformly distributed transverse force along its length. The beam has fixed geometry so we are not optimizing its shape, rather we are simply solving a discretization of the Euler-Bernoulli beam bending equations using GP.

```python
"""
A simple beam example with fixed geometry. Solves the discretized
Euler-Bernoulli beam equations for a constant distributed load
"""
import numpy as np
from gpkit import Variable, VectorVariable, Model, units
from gpkit.small_scripts import mag


class Beam(Model):
    """Discretization of the Euler beam equations for a distributed load.

    Arguments
    ---------
    N : int
        Number of finite elements that compose the beam.
    L : float
        [m] Length of beam.
    EI : float
        [N m^2] Elastic modulus times cross-section's area moment of inertia.
    q : float or N-vector of floats
        [N/m] Loading density: can be specified as constants or as an array.
    """
    def __init__(self, N=4, **kwargs):
        EI = Variable("EI", 1e4, "N*m^2")
        dx = Variable("dx", "m", "Length of an element")
        L = Variable("L", 5, "m", "Overall beam length")
        q = VectorVariable(N, "q", 100*np.ones(N), "N/m",
                           "Distributed load at each point")
        V = VectorVariable(N, "V", "N", "Internal shear")
        V_tip = Variable("V_{tip}", 0, "N", "Tip loading")
        M = VectorVariable(N, "M", "N*m", "Internal moment")
        M_tip = Variable("M_{tip}", 0, "N*m", "Tip moment")
        th = VectorVariable(N, "\\theta", "-", "Slope")
        th_base = Variable("\\theta_{base}", 0, "-", "Base angle")
        w = VectorVariable(N, "w", "m", "Displacement")
        w_base = Variable("w_{base}", 0, "m", "Base deflection")
        # below: trapezoidal integration to form a piecewise-linear
        #        approximation of loading, shear, and so on
        # shear and moment increase from tip to base (left > right)
        shear_eq = (V >= V.right + 0.5*dx*(q + q.right))
        shear_eq[-1] = (V[-1] >= V_tip)  # tip boundary condition
        moment_eq = (M >= M.right + 0.5*dx*(V + V.right))
        moment_eq[-1] = (M[-1] >= M_tip)
        # slope and displacement increase from base to tip (right > left)
        theta_eq = (th >= th.left + 0.5*dx*(M + M.left)/EI)
        theta_eq[0] = (th[0] >= th_base)  # base boundary condition
        displ_eq = (w >= w.left + 0.5*dx*(th + th.left))
        displ_eq[0] = (w[0] >= w_base)
        # minimize tip displacement (the last w)
        Model.__init__(self, w[-1],
                       [shear_eq, moment_eq, theta_eq, displ_eq,
                        L == (N-1)*dx], **kwargs)


b = Beam(N=6, substitutions={"L": 6, "EI": 1.1e4, "q": 110*np.ones(10)})
b.zero_lower_unbounded_variables()
sol = b.solve(verbosity=1)
w_gp = sol("w")  # deflection along beam

L, EI, q = sol("L"), sol("EI"), sol("q")
x = np.linspace(0, mag(L), len(q))*units.m  # position along beam
q = q[0]  # assume uniform loading for the check below
w_exact = q/(24.*EI) * x**2 * (x**2 - 4*L*x + 6*L**2)  # analytic soln

assert max(abs(w_gp - w_exact)) <= 1e-2*units.m

PLOT = False
if PLOT:
    import matplotlib.pyplot as plt
    x_exact = np.linspace(0, L, 1000)
    w_exact = q/(24.*EI) * x_exact**2 * (x_exact**2 - 4*L*x_exact + 6*L**2)
    plt.plot(x, w_gp, color='red', linestyle='solid', marker='^',
             markersize=8)
    plt.plot(x_exact, w_exact, color='blue', linestyle='dashed')
    plt.xlabel('x [m]')
    plt.ylabel('Deflection [m]')
    plt.axis('equal')
    plt.legend(['GP solution', 'Analytical solution'])
    plt.show()
```

The output is

```
Cost
----
 1.62 [m]

Free Variables
--------------
          dx : 1.2                                          [m]   Length of an element
     \vec{M} : [ 1.98e+03  1.27e+03  713       317      ... ]  [N*m] Internal moment
     \vec{V} : [ 660       528       396       264      ... ]  [N]   Internal shear
\vec{\theta} : [  -        0.177     0.285     0.341    ... ]        Slope
     \vec{w} : [  -        0.106     0.384     0.759    ... ]  [m]   Displacement

Constants
```

```
---------
          EI : 1.1e+04                                          [N*m**2]
           L : 6                                                [m]      Overall beam length
      M_{tip} : 0                                               [N*m]    Tip moment
      V_{tip} : 0                                               [N]      Tip loading
\theta_{base} : 0                                                        Base angle
     w_{base} : 0                                               [m]      Base deflection
      \vec{M} : [  -         -         -         -       ... ]  [N*m]    Internal moment
      \vec{V} : [  -         -         -         -       ... ]  [N]      Internal shear
 \vec{\theta} : [ 0          -         -         -       ... ]           Slope
      \vec{q} : [ 110       110       110       110      ... ]  [N/m]    Distributed load at eac
      \vec{w} : [ 0          -         -         -       ... ]  [m]      Displacement

Sensitivities
-------------
      L : 4                                              Overall beam length
\vec{q} : [ 0.0072    0.0416    0.118     0.234    ... ] Distributed load at each point
     EI : -1
```

By plotting the deflection, we can see that the agreement between the analytical solution and the GP solution is good.

CHAPTER 8

Glossary

For an alphabetical listing of all commands, check out the genindex

8.1 gpkit package

8.1.1 Subpackages

gpkit.constraints package

Submodules

gpkit.constraints.array module

gpkit.constraints.costed module

gpkit.constraints.linked module

gpkit.constraints.model module

gpkit.constraints.prog_factories module

gpkit.constraints.set module

gpkit.constraints.signomial_program module

gpkit.constraints.single_equation module

gpkit.constraints.tight module

Module contents

gpkit.interactive package

Submodules

gpkit.interactive.chartjs module

gpkit.interactive.plotting module

gpkit.interactive.ractor module

gpkit.interactive.sensitivity_map module

gpkit.interactive.widgets module

Module contents

gpkit.nomials package

Submodules

gpkit.nomials.array module

gpkit.nomials.data module

gpkit.nomials.nomial_core module

CHAPTER 9

Citing GPkit

If you use GPkit, please cite it with the following bibtex:

```
@Misc{gpkit,
      author={Edward Burnell and Warren Hoburg},
      title={GPkit software for geometric programming},
      howpublished={\url{https://github.com/hoburg/gpkit}},
      year={2015},
      note={Version 0.4.0}
     }
```

CHAPTER 10

Acknowledgements

We thank the following contributors for helping to improve GPkit:

- Marshall Galbraith for setting up continuous integration.
- Stephen Boyd for inspiration and suggestions.

CHAPTER 11

Release Notes

This page lists the changes made in each point version of gpkit.

11.1 Version 0.3

- Integrated GP and SP creation under the Model class
- Improved and simplified under-the-hood internals of GPs and SPs
- New experimental SP heuristic
- Improved test coverage
- Handles vectors which are partially constants, partially free
- Simplified interaction with Model objects and made it more pythonic
- Added SP "step" method to allow single-stepping through an SP
- Isolated and corrected some solver-specific behavior
- Fully allowed substitutions of variables for 0 (commit 4631255)
- Use "with" to create a signomials environment (commit cd8d581)
- Continuous integration improvements, thanks @galbramc !
- Not counting subpackages, went from 2200 to 2400 lines of code (additions were mostly longer error messages) and from 650 to 1050 lines of docstrings and comments.
- Add automatic feasibility-analysis methods to Model and GP
- Simplified solver logging and printing, making it easier to access solver output.

11.2 Version 0.2

- Various bug fixes
- Python 3 compatibility
- Added signomial programming support (alpha quality, may be wrong)
- Added composite objectives
- Parallelized sweeping

- Better table printing
- Linked sweep variables
- Better error messages
- Closest feasible point capability
- Improved install process (no longer requires ctypesgen; auto-detects MOSEK version)
- Added examples: wind turbine, modular GP, examples from 1967 book, maintenance (part replacement)
- Documentation grew by ~70%
- Added Advanced Commands section to documentation
- Many additional unit tests (more than doubled testing lines of code)