gpkit Documentation

Release 0.3.3

MIT Department of Aeronautics and Astronautics

September 29, 2015

Contents

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 special type of constrained non-linear optimization problem.

A GP is made up of special types of functions called *monomials* and *posynomials*. In the context of GP, a monomial is defined as:

$$f(x) = cx_1^{a_1}x_2^{a_2}...x_n^{a_n}$$

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

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

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:

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

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

$$
llt \quad \text{minimize} \\
x^{-1}y^{-1/2}z^{-1} + 2.3xz + 4xyz \\
\text{subject to} \\
(1/3)x^{-2}y^{-2} + (4/3)y^{1/2}z^{-1} \leq 1
$$

$$x + 2y + 3z \leq 1$$

$$(1/2)xy = 1$$

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.

CHAPTER 2

GPkit Overview

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

The primary goal of GPkit is to make it easy to create share and explore Geometric Programming models. Being fast and mathematically correct tend to align well with this goal.

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 look at 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.

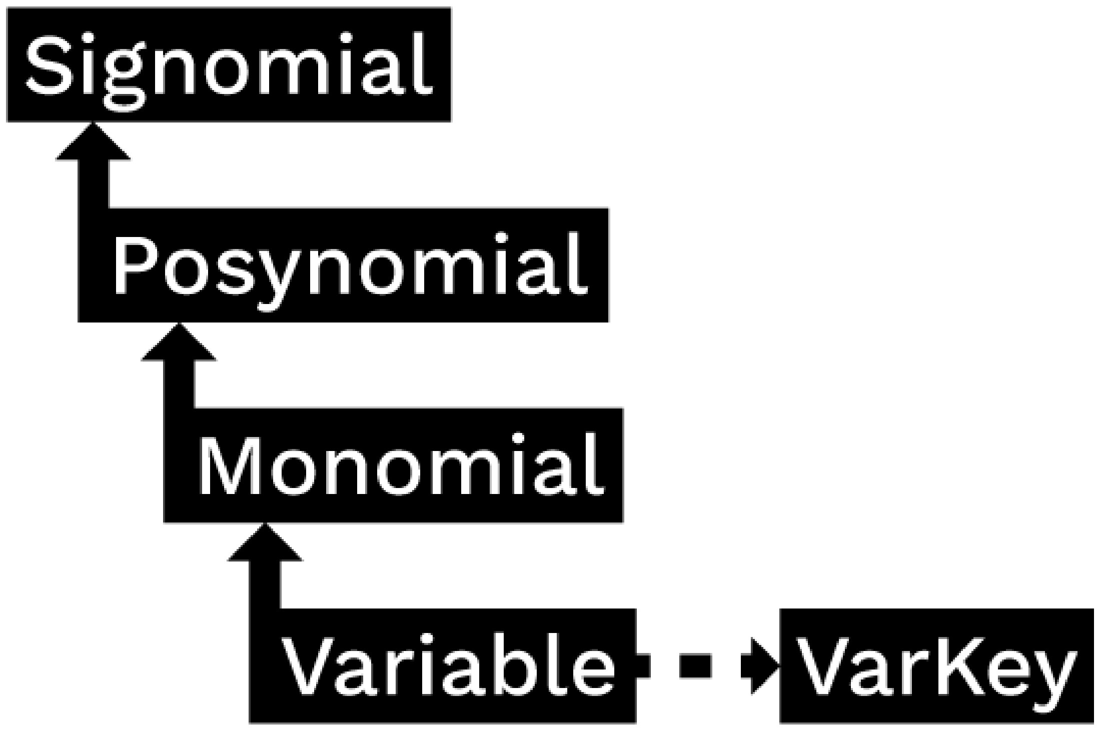
2.2 Substitution

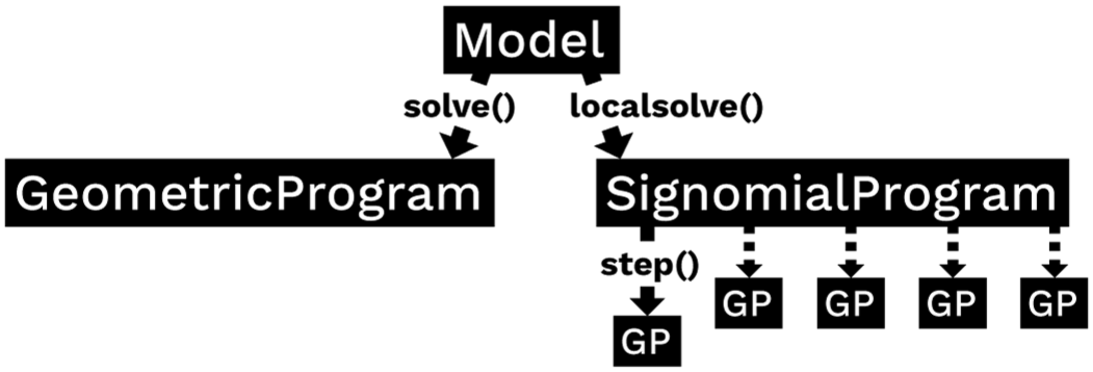
The `Varkey` object in the graph above is not a algebraic expression, but what GPkit uses as a variable's "name". It carries the LaTeX representation of a variable and its units, as well as any other information the user wishes to associate with a variable. The use of `VarKeys` as opposed to numeric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

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 any bugs during installation, please email `gpkit` at `mit.edu`.

3.1 Mac OS X

3.1.1 1. Install Python and build dependencies

- Install the Python 2.7 version of Anaconda. - Check that Anaconda is installed: in a Terminal window, run `python` and check that the version string it prints while starting includes "Anaconda".
  - If it does not, check that the Anaconda location in `.profile` in your home directory (you can run `vim ~/.profile` to read it) corresponds to the location of your Anaconda folder; if it doesn't, move the Anaconda folder there, and check again in the `python` startup header.
- If you don't want to install Anaconda, you'll need gcc, pip, numpy, and scipy, and may find iPython Notebook useful as a modeling environment.
- If `which gcc` does not return anything, install the Apple Command Line Tools.
- *Optional:* to install gpkit into an isolated python environment you can create a new conda virtual environment with `conda create -n gpkit anaconda` and activate it with `source activate gpkit`.
- Run `pip install ctypesgen --pre` in the Terminal if you want to use a MOSEK solver.

3.1.2 2. Install either the MOSEK or CVXOPT GP solvers

- **Download CVXOPT, then:**
  - Read the official instructions and requirements
  - In the Terminal, navigate to the `cvxopt` folder
  - Run `python setup.py install`
- **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/`

3.1.3 3. Install GPkit

- Run `pip install gpkit` at the command line.
- Run `python -c "import gpkit.tests; gpkit.tests.run()"`
- If you want units support, install pint with `pip install pint`.

3.2 Linux

3.2.1 1. Install either the MOSEK or CVXOPT GP solvers

- **Download CVXOPT, then:**
  - Read the official instructions and requirements
  - In a terminal, navigate to the `cvxopt` folder
  - Run `python setup.py install`
- **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/`

3.2.2 2. Install GPkit

- _Optional:_ to install gpkit into an isolated python environment, install virtualenv, run `virtualenv $DESTINATION_DIR` then activate it with `source activate $DESTINATION_DIR/bin`.
- Run `pip install ctypesgen --pre` at the command line if you want to use a MOSEK solver.
- Run `pip install gpkit` at the command line.
- Run `python -c "import gpkit.tests; gpkit.tests.run()"`
- If you want units support, install pint with `pip install pint`.
- You may find iPython Notebook to be useful modeling environment.

3.3 Windows

3.3.1 1. Install Python dependencies

- Install the Python 2.7 version of Anaconda.
- If you don't want to install Anaconda, you'll need gcc, pip, numpy, and scipy, and may find iPython Notebook useful as a modeling environment.
- *Optional:* to install gpkit into an isolated python environment you can create a new conda virtual environment with `conda create -n gpkit anaconda` and activate it with `source activate gpkit`.

- Run `pip install ctypesgen --pre` at an Anaconda Command Prompt if you want to use a MOSEK solver.

3.3.2 2. Install either the MOSEK or CVXOPT GP solvers

- Download CVXOPT, then follow these steps to install a linear algebra library
- **Download MOSEK, then:**
  - Follow these steps for Windows.
  - Request an academic license file and put it in `~/mosek/`

3.3.3 3. Install GPkit

- Run `pip install gpkit` at an Anaconda Command Prompt.
- Run `python -c "import gpkit.tests; gpkit.tests.run()"`
- If you want units support, install pint with `pip install pint`.

CHAPTER 4

Getting Started

GPkit is a Python package. We assume basic familiarity with Python. If you are new to Python take a look at Learn Python.

GPkit is also a command line tool. This means that you need to be in the terminal (OS X/Linux) or command prompt (Windows) to use it. If you are not familiar with working in the command line, check out this Learn Code the Hard Way tutorial.

The first thing to do is install GPkit . Once you have done this, you can start using GPkit in 3 easy steps:

1. Open your command line interface (terminal/Command Prompt).
2. Open a Python interpreter. This can be done by typing `python` (or `ipython` if installed).
3. Type `import gpkit`.

After doing this, your command line will look something like the following:

```
$ python
>>> import gpkit
>>>
```

From here, you can use GPkit commands to formulate and solve geometric programs. To learn how, see Basic Commands.

4.1 Writing GPkit Scripts

Another way to use GPkit is to write a script and save it as a .py file. To run this file (e.g. `myscript.py`), type the following in your command line:

```
$ python myscript.py
```

Again, `ipython` will also work here.

CHAPTER 5

Basic Commands

5.1 Importing Modules

The first thing to do when using GPkit is to import the classes and modules you will need. For example,

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

5.2 Declaring Variables

Instances of the `Variable` class represent scalar decision 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).

5.2.1 Decision Variables

```
# 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')
```

Note: make sure you have imported the class `Variable` beforehand.

5.2.2 Fixed Variables

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

```
# Declare \rho equal to 1.225 kg/m^3.
# NOTE: starting a Python string with 'r' makes the backslashes literal,
#       which is useful for LaTeX strings.
rho = Variable(r'\rho', 1.225, 'kg/m^3', 'Density of air at sea level')
```

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

```
#Declare pi equal to 3.14
pi = Variable(r'\pi', 3.14)
```

5.2.3 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", "3-D Position")
```

5.3 Creating Monomials and Posynomials

Monomial and posynomial expressions can be created using mathematical operations on variables. This is implemented under-the-hood using operator overloading in Python.

```
# 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
```

5.4 Declaring Constraints

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

5.5 Declaring Objective Functions

To declare an objective function, assign a Posynomial (or Monomial) to a variable name, such as `objective`.

```
objective = 1/(x*y*z)
```

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

5.6 Formulating a Model

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

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

5.7 Solving the Model

```
sol = gp.solve()
```

5.8 Printing Results

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

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

CHAPTER 6

Advanced Commands

6.1 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, PosyArray
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 = PosyArray(m.signomials)/feas["overall"]
m.solve()

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

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

6.2 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 less-than side of each constraint would have on the overall objective (called the "dual solution", "shadow prices", or "posynomial sensitivities").

6.2.1 Using variable sensitivities

GPkit uses this dual solution to compute the sensitivities of each variable, which can be accessed most easily using a GPSolutionArray's `senssubinto()` 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.senssubinto(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.senssubinto(x_squared_min) == 2
```

6.2.2 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.

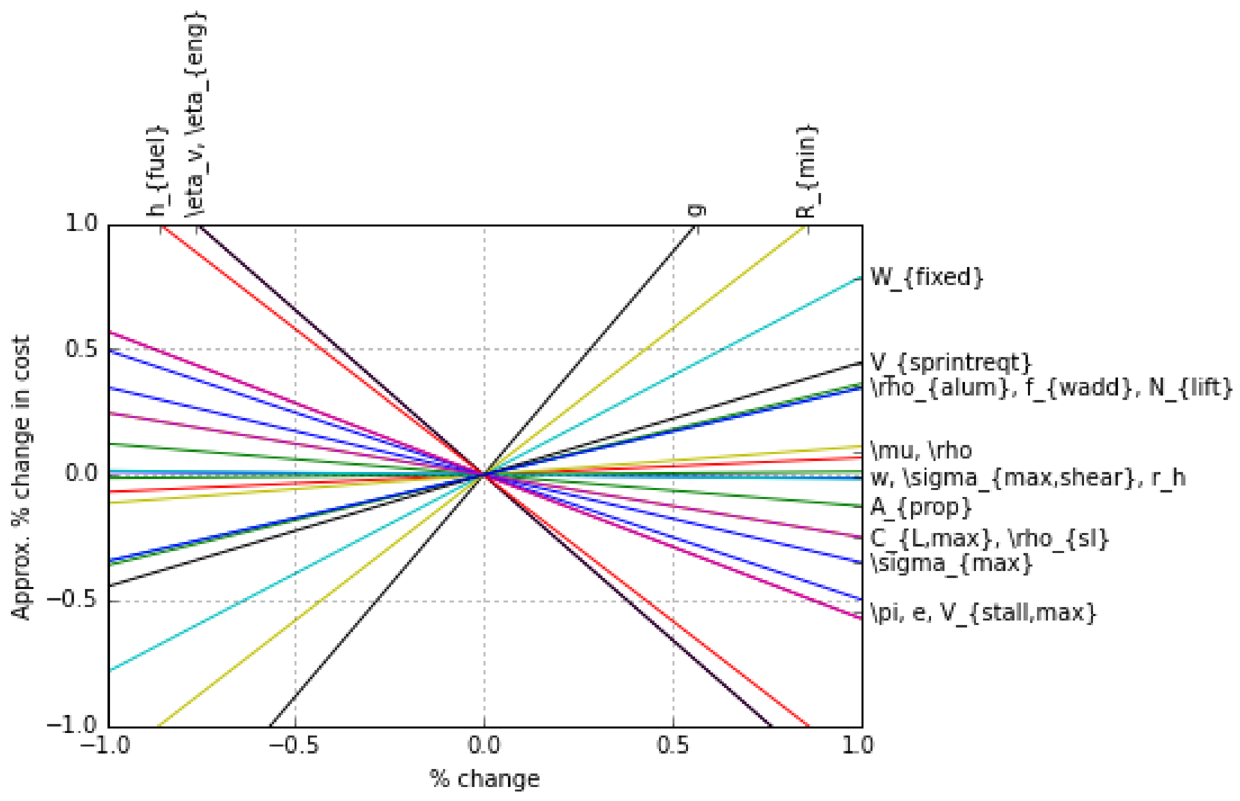
6.3 Substitutions

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

6.3.1 Substituting into Posynomials, PosyArrays, and GPs

The examples below all use Posynomials and PosyArrays, 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.

6.3.2 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 PosyArray.

6.3.3 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).

6.3.4 Substituting with replacement

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

6.3.5 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 PosyArray. 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`.

6.4 Sweeps

6.4.1 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))`.

6.4.2 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 $y \forall y_i \in \{1, 2\}$.

6.4.3 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.

6.4.4 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.

6.4.5 Example Usage

```python
# 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)
```

6.5 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.

6.5.1 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.

6.6 Signomial Programming

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

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

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 the signomial problem affects its solve time in a nuanced way: `gpkit.SP(x, [x >= 0.1, x+y >= 1, y <= 0.1]).localsolve()` takes about four times as many iterations to solve as `gpkit.SP(x, [x >= 1-y, y <= 0.1]).localsolve()`, despite the two formulations being arithmetically equivalent.

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.

6.6.1 Example Usage

```
"""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.

CHAPTER 7

Examples

7.1 iPython Notebook Examples

More examples, including some in-depth and experimental models, can be seen on nbviewer.

7.2 A Trivial GP

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

```
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.

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

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):

```
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
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.

The output is

```
Using solver 'cvxopt'
Solving for 9 variables.
Solving took 0.0492 seconds.

Cost
----
 255
```

```
Free Variables
--------------
  A : 12.7               Aspect ratio
C_D : 0.0231             Drag coefficient
C_L : 0.6512             Lift coefficient
C_f : 0.003857           Skin friction coefficient
 Re : 2.598e+06          Reynolds number
  S : 12.08      [m**2]  Wing planform area
  V : 38.55      [m/s]   Cruise velocity
  W : 7189       [N]     Total aircraft weight
W_w : 2249       [N]     Wing weight

Constants
---------
          (CDA_0) : 0.0306    [m**2]    Fuselage drag area
        C_{L,max} : 2                   Maximum C_L, flaps down
         N_{lift} : 2.5                 Ultimate load factor
          V_{min} : 22        [m/s]     Desired landing speed
              W_0 : 4940      [N]       Aircraft weight excluding wing
\frac{S_{wet}}{S} : 2.05                Wetted area ratio
              \mu : 1.78e-05  [kg/m/s]  Viscosity of air
             \rho : 1.23      [kg/m**3] Density of air
             \tau : 0.12                Airfoil thickness-to-chord ratio
             cww1 : 45.42     [N/m**2]  Wing weight area factor
             cww2 : 8.71e-05  [1/m]     Wing weight bending factor
                e : 0.96                Oswald efficiency factor
                k : 1.2                 Form factor

Sensitivities
-------------
          (CDA_0) : 0.1097  Fuselage drag area
        C_{L,max} : -0.1307 Maximum C_L, flaps down
         N_{lift} : 0.2923  Ultimate load factor
          V_{min} : -0.2614 Desired landing speed
              W_0 : 0.9953  Aircraft weight excluding wing
\frac{S_{wet}}{S} : 0.4108  Wetted area ratio
              \mu : 0.08217 Viscosity of air
             \rho : -0.1718 Density of air
             \tau : -0.2923 Airfoil thickness-to-chord ratio
             cww1 : 0.09428 Wing weight area factor
             cww2 : 0.2923  Wing weight bending factor
                e : -0.4795 Oswald efficiency factor
                k : 0.4108  Form factor
```

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.

```
"""
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.shortcuts import *
```

```
def beam(N=10, L=5., EI=1E4, P=100):

    dx = Var("dx", L/(N-1), units="m")
    EI = Var("EI", EI, units="N*m^2")

    p = Vec(N, "p", units="N/m", label="Distributed load")
    p = p.sub(p, P*np.ones(N))

    V  = Vec(N, "V", units="N", label="Internal shear")
    M  = Vec(N, "M", units="N*m", label="Internal moment")
    th = Vec(N, "th", units="-", label="Slope")
    w  = Vec(N, "w", units="m", label="Displacement")

    eps = 1E-16 #an arbitrarily small positive number

    substitutions = {var: eps for var in [V[-1], M[-1], th[0], w[0]]}

    objective = w[-1]

    constraints = [V.left[1:N]     >= V[1:N]    + 0.5*dx*(p.left[1:N]    + p[1:N]),
                   M.left[1:N]     >= M[1:N]    + 0.5*dx*(V.left[1:N]    + V[1:N]),
                   th.right[0:N-1] >= th[0:N-1] + 0.5*dx*(M.right[0:N-1] + M[0:N-1])/EI,
                   w.right[0:N-1]  >= w[0:N-1]  + 0.5*dx*(th.right[0:N-1]+ th[0:N-1])
                  ]

    return Model(objective, constraints, substitutions)


N = 10 #  [-] grid size
L = 5. #   [m] beam length
EI = 1E4 # [N*m^2] elastic modulus * area moment of inertia
P = 100 #  [N/m] magnitude of distributed load

m = beam(N, L, EI, P)
sol = m.solve(verbosity=1)

x = np.linspace(0, L, N) # position along beam
w_gp = sol("w") # deflection along beam
w_exact =  P/(24.*EI)* x**2 * (x**2  - 4*L*x + 6*L**2) # analytical soln

assert max(abs(w_gp - w_exact)) <= 1e-2

PLOT = False
if PLOT:
    import matplotlib.pyplot as plt
    x_exact = np.linspace(0, L, 1000)
    w_exact =  P/(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
----
 0.7813 [m]

Free Variables
--------------
 M : [ 1.25e+03  988       756       556       ... ]  [N*m] Internal moment
 V : [ 500       444       389       333       ... ]  [N]   Internal shear
th : [  -        0.0622    0.111     0.147     ... ]        Slope
 w : [  -        0.0173    0.0653    0.137     ... ]  [m]   Displacement

Constants
---------
EI : 1e+04                                              [N*m**2]
dx : 0.5556                                             [m]
 M : [  -         -         -         -        ... ]  [N*m]    Internal moment
 V : [  -         -         -         -        ... ]  [N]      Internal shear
th : [ 1e-16      -         -         -        ... ]           Slope
 w : [ 1e-16      -         -         -        ... ]  [m]      Displacement

Sensitivities
-------------
EI : -1
dx : 4
```

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 The GPkit Package

8.2 Subpackages

8.2.1 gpkit.interactive

gpkit.interactive.plotting module

gpkit.interactive.pretty_print module

gpkit.interactive.printing module

gpkit.interactive.ractor module

gpkit.interactive.widget module

8.3 Submodules

8.3.1 gpkit.model

8.3.2 gpkit.solution_array module

8.3.3 gpkit.geometric_program

8.3.4 gpkit.signomial_program

8.3.5 gpkit.nomials

8.3.6 gpkit.posyarray

8.3.7 gpkit.small_classes

8.3.8 gpkit.small_scripts

8.3.9 gpkit.substitution

8.3.10 gpkit.tools module

8.3.11 gpkit.variables module

8.3.12 gpkit.varkey module

CHAPTER 9

Citing GPkit

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

```
@Misc{gpkit,
      author={MIT Department of Aeronautics and Astronautics},
      title={GPkit},
      howpublished={\url{https://github.com/convexopt/gpkit}},
      year={2015},
      note={Version 0.3.3}
     }
```

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)