

---

# **gpkIt Documentation**

***Release 0.3.1***

**MIT Department of Aeronautics and Astronautics**

August 17, 2015



<b>1</b>	<b>Geometric Programming 101</b>	<b>3</b>
1.1	What is a GP? . . . . .	3
1.2	Why are GPs special? . . . . .	4
1.3	What are Signomials / Signomial Programs? . . . . .	4
1.4	Where can I learn more? . . . . .	4
<b>2</b>	<b>GPkit Overview</b>	<b>5</b>
2.1	Symbolic expressions . . . . .	5
2.2	Substitution . . . . .	5
2.3	Model objects . . . . .	5
<b>3</b>	<b>Installation Instructions</b>	<b>9</b>
3.1	Mac OS X . . . . .	9
3.2	Linux . . . . .	10
3.3	Windows . . . . .	10
<b>4</b>	<b>Getting Started</b>	<b>13</b>
4.1	Writing GPkit Scripts . . . . .	13
<b>5</b>	<b>Basic Commands</b>	<b>15</b>
5.1	Importing Modules . . . . .	15
5.2	Declaring Variables . . . . .	15
5.3	Creating Monomials and Posynomials . . . . .	16
5.4	Declaring Constraints . . . . .	16
5.5	Declaring Objective Functions . . . . .	16
5.6	Formulating a Model . . . . .	17
5.7	Solving the Model . . . . .	17
5.8	Printing Results . . . . .	17
<b>6</b>	<b>Advanced Commands</b>	<b>19</b>
6.1	Feasibility Analysis . . . . .	19
6.2	Sensitivities and dual variables . . . . .	19
6.3	Substitutions . . . . .	20
6.4	Sweeps . . . . .	22
6.5	Composite Objectives . . . . .	23
6.6	Signomial Programming . . . . .	24
<b>7</b>	<b>Examples</b>	<b>27</b>
7.1	iPython Notebook Examples . . . . .	27

7.2	A Trivial GP . . . . .	27
7.3	Maximizing the Volume of a Box . . . . .	27
7.4	Water Tank . . . . .	29
7.5	Simple Wing . . . . .	29
<b>8</b>	<b>Glossary</b>	<b>33</b>
8.1	The GPkit Package . . . . .	33
8.2	Subpackages . . . . .	34
8.3	Submodules . . . . .	34
<b>9</b>	<b>Citing GPkit</b>	<b>47</b>
<b>10</b>	<b>Acknowledgements</b>	<b>49</b>
<b>11</b>	<b>Release Notes</b>	<b>51</b>
11.1	Version 0.3 . . . . .	51
11.2	Version 0.2 . . . . .	51
	<b>Python Module Index</b>	<b>53</b>



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



---

## 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:

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

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

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

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

$$g(x) = \sum_{k=1}^K c_k x_1^{a_{1k}} x_2^{a_{2k}} \dots x_n^{a_{nk}}$$

For example, the expressions

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

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:

$$\begin{aligned} & \text{lt} \quad \text{minimize} \\ & \quad g_0(x) \end{aligned}$$

$$\begin{aligned} & \text{subject to} \\ & \quad f_i(x) = 1, \\ & \quad i = 1, \dots, m \end{aligned}$$

$$\begin{aligned} & \quad g_i(x) \leq 1, \\ & \quad i = 1, \dots, n \end{aligned}$$

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

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

## 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.



---

## 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.

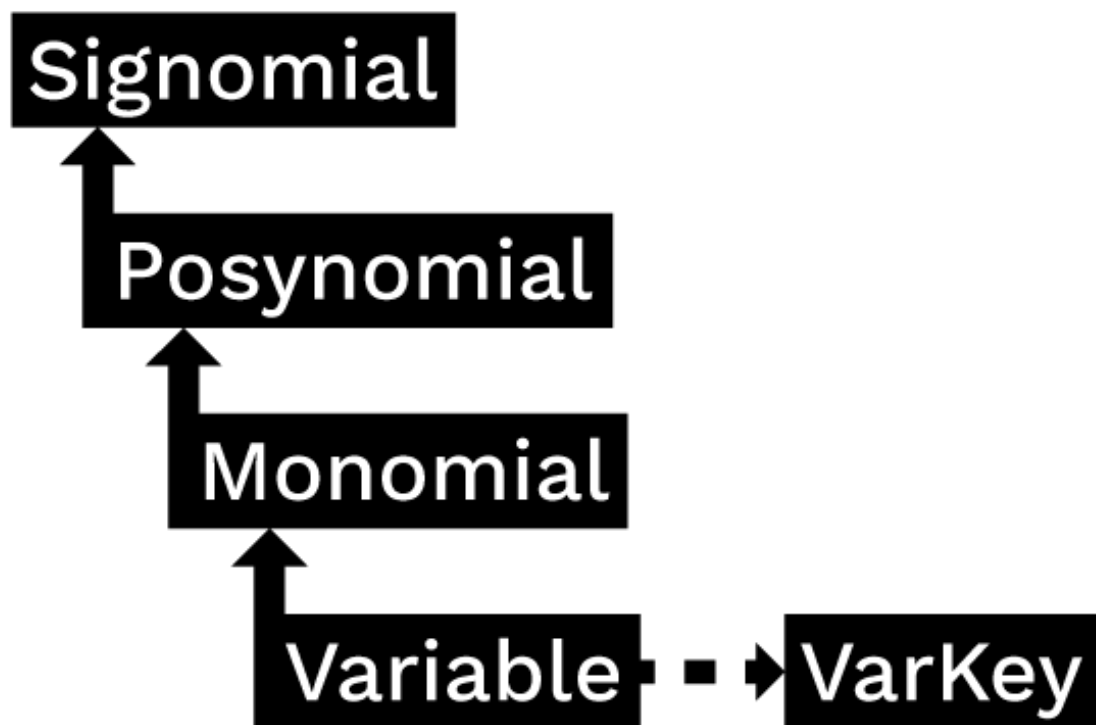
### 2.2 Substitution

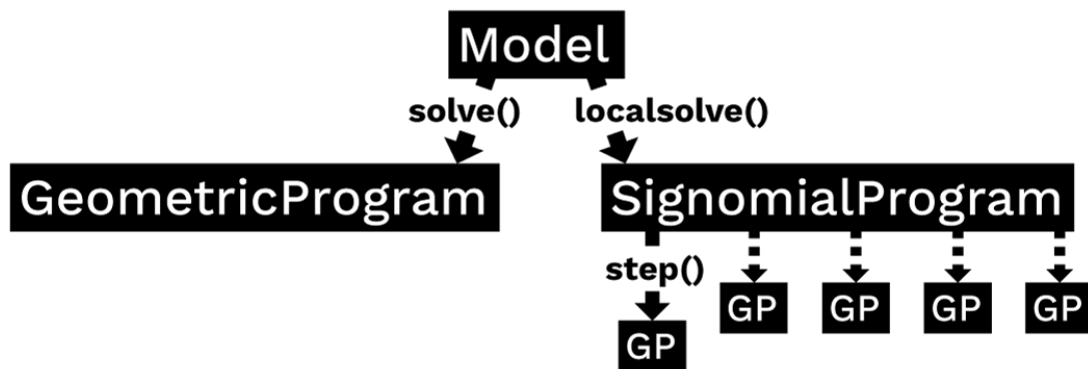
The `Varkey` object in the graph above is not a algebraic expression, but what GPkit uses as a variable’s “name”. It carries the LaTeX representation of a variable and its units, as well as any other information the user wishes to associate with a variable. The use of `VarKeys` as opposed to numeric 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.







---

## Installation Instructions

---

If you encounter any bugs during installation, please email [gpkit@mit.edu](mailto:gpkit@mit.edu).

### 3.1 Mac OS X

#### 3.1.1 1. Install Python and build 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.
- 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`.





---

## 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.



---

## 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))
```



---

## 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 gpkits 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 RuntimeError!
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

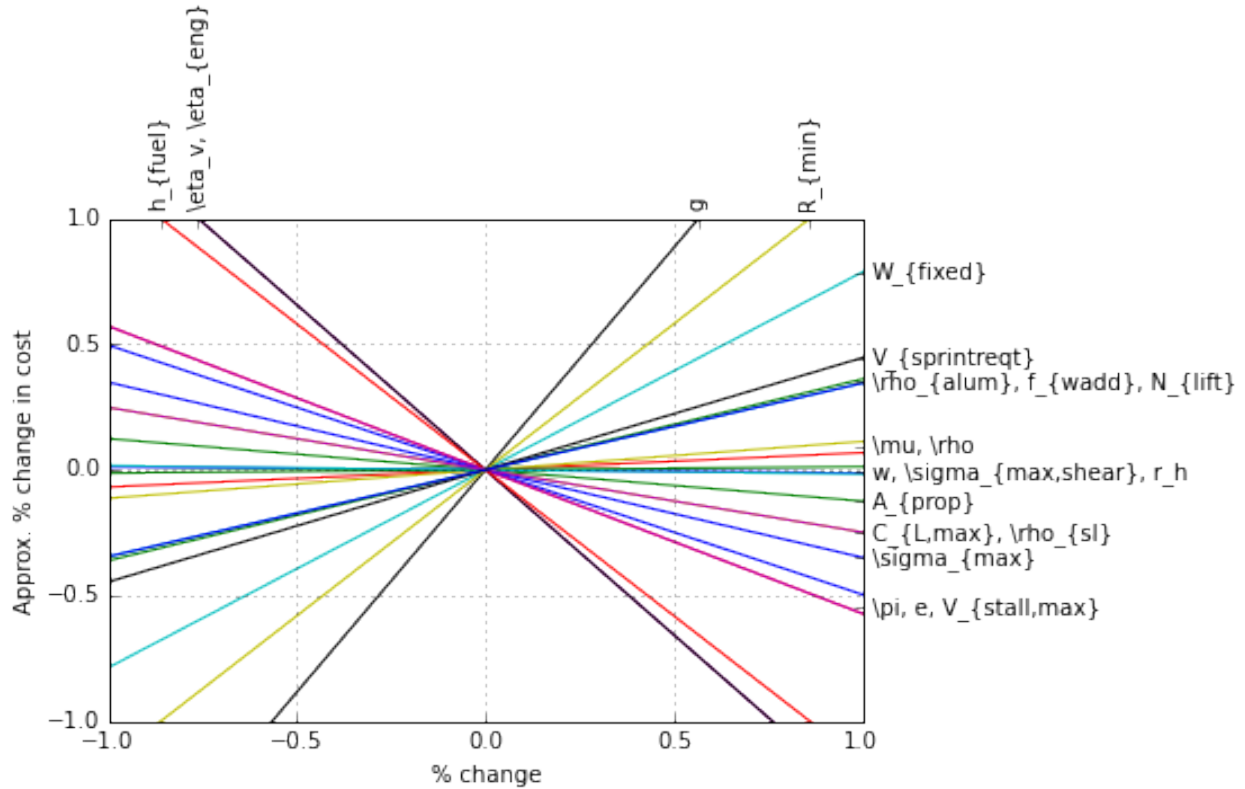
Substitutions are a general-purpose way to change every instance of one variable into either a number or another variable.

### 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-substituted 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 is 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 sweep 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

```
# 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:

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

where  $i \in 0 \dots n - 1$  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:

$$\begin{aligned} &lt; \text{minimize} \\ &\quad g_0(x) \\ & \\ &\text{subject to} \\ &\quad f_i(x) = 1, \\ &\quad i = 1, \dots, m \\ & \\ &\quad g_i(x) - h_i(x) \leq 1, \\ &\quad i = 1, \dots, n \end{aligned}$$

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=0)
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.



---

## 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 gpykit 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
sol = m.solve(verbosity=0)

# Print results table
print sol.table()
```

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=0)
print sol.table()
```

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.

```
from gpkit.shortcuts import *
import numpy as np
```

```

# Define the constants in the problem
CDA0 = Var('(CDA_0)', 0.0306, 'm^2', label='Fuselage drag area')
C_Lmax = Var('C_{L,max}', 2.0, label='Maximum C_L, flaps down')
e = Var('e', 0.96, label='Oswald efficiency factor')
k = Var('k', 1.2, label='Form factor')
mu = Var('\\\\mu', 1.78E-5, 'kg/m/s', label='Viscosity of air')
N_lift = Var('N_{lift}', 2.5, label='Ultimate load factor')
rho = Var('\\\\rho', 1.23, 'kg/m^3', label='Density of air')
Sw_S = Var('\\\\frac{S_{wet}}{S}', 2.05, label='Wetted area ratio')
tau = Var('\\\\tau', 0.12, label='Airfoil thickness-to-chord ratio')
V_min = Var('V_{min}', 22, 'm/s', label='Desired landing speed')
W_0 = Var('W_0', 4940, 'N', label='Aircraft weight excluding wing')
cww1 = Var('cww1', 45.42, 'N/m^2', label='Wing weight area factor')
cww2 = Var('cww2', 8.71E-5, '1/m', label='Wing weight bending factor')

# Define decision variables
A = Var('A', label='Aspect ratio')
C_D = Var('C_D', label='Drag coefficient')
C_L = Var('C_L', label='Lift coefficient')
C_f = Var('C_f', label='Skin friction coefficient')
S = Var('S', 'm^2', label='Wing planform area')
Re = Var('Re', label='Reynolds number')
W = Var('Re', 'N', label='Total aircraft weight')
W_w = Var('W_w', 'N', label='Wing weight')
V = Var('V', 'm/s', label='Cruise velocity')

# Define objective function
objective = 0.5 * rho * V**2 * C_D * S

# Define constraints
constraints = [C_f * Re**0.2 >= 0.074,
               C_D >= CDA0 / S + k * C_f * Sw_S + C_L**2 / (np.pi * A * e),
               0.5 * rho * V**2 * C_L * S >= W,
               W >= W_0 + W_w,
               W_w >= (cww1*S +
                       cww2 * N_lift * A**1.5 * (W_0 * W * S)**0.5 / tau),
               0.5 * rho * V_min**2 * C_Lmax * S >= W,
               Re == rho * V * (S/A)**0.5 / mu]

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

# Solve the Model
sol = m.solve()

```

The output is

```

Using solver 'cvxopt'
Solving for 9 variables.
Solving took 0.0487 seconds.

Cost
----
255 [kg*m/s**2]

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
Re : 7189             [N]   Total aircraft weight
S : 12.08             [m**2] Wing planform area
V : 38.55             [m/s] Cruise velocity
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

```



---

## Glossary

---

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

### 8.1 The GPkit Package

Lightweight GP Modeling Package

For examples please see the examples folder.

#### 8.1.1 Requirements

numpy MOSEK or CVXOPT scipy(optional): for complete sparse matrix support sympy(optional): for latex printing in iPython Notebook

#### 8.1.2 Attributes

**settings** [dict] Contains settings loaded from `./env/settings`

**class** `gpkit.SignomialsEnabled`

Bases: `object`

Class to put up and tear down signomial support in an instance of GPkit.

```
>>> import gpkit
>>> x = gpkit.Variable("x")
>>> y = gpkit.Variable("y", 0.1)
>>> with SignomialsEnabled():
>>>     constraints = [x >= 1-y]
>>> gpkit.Model(x, constraints).localsolve()
```

`gpkit.disable_signomials()`

Disables signomial support in a particular instance of GPkit.

`gpkit.disable_units()`

Disables units support in a particular instance of GPkit.

Posynomials created after calling this are incompatible with those created before.

If gpkit is imported multiple times, this needs to be run each time.

**The correct way to call this is:** `import gpkit gpkit.disable_units()`

The following will *not* have the intended effect: `from gpkIt import disable_units disable_units()`

`gpkIt.enable_signomials()`

Enables signomial support in a particular instance of GPkit.

`gpkIt.enable_units()`

Enables units support in a particular instance of GPkit.

Posynomials created after calling this are incompatible with those created before.

If gpkIt is imported multiple times, this needs to be run each time.

## 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

Module for creating Model instances.

#### Example

```
>>> gp = gpkIt.Model(cost, constraints, substitutions)
```

```
class gpkIt.model.Model(cost=None, constraints=None, substitutions=None, *args,  
                        **kwargs)
```

Bases: object

Symbolic representation of an optimization problem.

The Model class is used both directly to create models with constants and sweeps, and indirectly inherited to create custom model classes.

**cost** [Signomial (optional)] If this is undeclared, the Model will get its cost and constraints from its “setup” method. This allows for easy inheritance.

**constraints** [list of Constraints (optional)] Defaults to an empty list.

**substitutions** [dict (optional)] This dictionary will be substituted into the problem before solving, and also allows the declaration of sweeps and linked sweeps.

**\*args, \*\*kwargs** : Passed to the setup method for inheritance.

*program* is set during a solve *solution* is set at the end of a solve

**allsubs**

All substitutions currently in the Model.

**constants**

All constants (non-sweep substitutions) currently in the Model.

**feasibility** (*search*=[*'overall'*, *'constraints'*, *'constants'*], *constvars*=None, *verbosity*=0)

Searches for feasible versions of the Model.

**search** [list of strings or string] The search(es) to perform. Details on each type below.

**constvars** [iterable] If declared, only constants in constvars will be changed. Otherwise, all constants can be changed in a constants search.

**verbosity** [int] If greater than 0, will print a report. Decrement by 1 and passed to solvers.

**feasibilities** [dict, float, or list] Has an entry for each search; if only one, returns that directly.

“**overall**” [float] The smallest number each constraint’s less-than side would have to be divided by to make the program feasible.

“**constraints**” [array of floats] Similar to “overall”, but contains a number for each constraint, and minimizes the product of those numbers.

“**constants**” [dict of varkeys: floats] A substitution dictionary that would make the program feasible, chosen to minimize the product of new\_values/old\_values.

```
>>> 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() # RuntimeError!
>>> 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()
```

**gp** (*verbosity*=2)

**localsolve** (*solver*=None, *verbosity*=2, *skipfailures*=True, *\*args*, *\*\*kwargs*)

Forms a SignomialProgram and attempts to locally solve it.

**solver** [string or function (optional)] If None, uses the default solver found in installation.

**verbosity** [int (optional)] If greater than 0 prints runtime messages. Is decremented by one and then passed to programs.

**skipfailures** [bool (optional)] If True, when a solve errors during a sweep, skip it.

*\*args*, *\*\*kwargs* : Passed to solver

**sol** [SolutionArray] See the SolutionArray documentation for details.

ValueError if called on a model without Signomials. RuntimeError if an error occurs in solving or parsing the solution.

**model\_nums** = defaultdict(<type 'int'>, {})

**signomials**

**signomials\_et\_al**

Get signomials, unsubbed, allsubs in one pass.

**solve** (*solver=None, verbosity=2, skipfailures=True, \*args, \*\*kwargs*)

Forms a GeometricProgram and attempts to solve it.

**solver** [string or function (optional)] If None, uses the default solver found in installation.

**verbosity** [int (optional)] If greater than 0 prints runtime messages. Is decremented by one and then passed to programs.

**skipfailures** [bool (optional)] If True, when a solve errors during a sweep, skip it.

**\*args, \*\*kwargs** : Passed to solver

**sol** [SolutionArray] See the SolutionArray documentation for details.

ValueError if called on a model with Signomials. RuntimeWarning if an error occurs in solving or parsing the solution.

**sp** (*verbosity=2*)

**unsubbed**

`gpkit.model.form_program(programType, signomials, verbosity=2)`

Generates a program and returns it and its solve function.

## 8.3.2 gpkit.solution\_array module

**class** `gpkit.solution_array.SolutionArray`

Bases: `gpkit.small_classes.DictOfLists`

A dictionary (of dictionaries) of lists, with convenience methods.

**cost** : array variables: dict of arrays sensitivities: dict containing:

monomials : array posynomials : array variables: dict of arrays

**localmodels** [PosyArray] Local power-law fits (small sensitivities are cut off)

```
>>> import gpkit
>>> import numpy as np
>>> x = gpkit.Variable("x")
>>> x_min = gpkit.Variable("x_{min}", 2)
>>> sol = gpkit.Model(x, [x >= x_min]).solve(verbosity=0)
>>>
>>> # VALUES
>>> values = [sol(x), sol.subinto(x), sol["variables"]["x"]]
>>> assert all(np.array(values) == 2)
>>>
>>> # SENSITIVITIES
>>> senss = [sol.sens(x_min), sol.senssubinto(x_min)]
>>> senss.append(sol["sensitivities"]["variables"]["x_{min}"])
>>> assert all(np.array(senss) == 1)
```

**getvars** (*\*args*)

**sens** (*p*)

**senssubinto** (*p*)

Returns array of each solution's sensitivity substituted into p

Returns only scalar values.



**subinto** (*p*)  
Returns PosyArray of each solution substituted into *p*.

**table** (*tables=['cost', 'freevariables', 'sweepvariables', 'constants', 'sensitivities'], fixedcols=True*)

`gpkit.solution_array.parse_result` (*result, constants, unsubbed, sweep={}, linkedsweep={}, freevar\_sensitivity\_tolerance=0.0001, localmodel\_sensitivity\_requirement=0.1*)

Parses a GP-like result dict into a SolutionArray-like dict.

`gpkit.solution_array.results_table` (*data, title, minval=0, printunits=True, fixedcols=True, varfmt='%s : ', valfmt='%-.4g ', vecfmt='%-8.3g'*)

Pretty string representation of a dict of VarKeys Iterable values are handled specially (partial printing)

**data:** dict whose keys are VarKey's data to represent in table

**title:** string **minval:** float

skip values with  $\text{all}(\text{abs}(\text{value})) < \text{minval}$

**printunits:** bool **fixedcols:** bool

if True, print rhs (val, units, label) in fixed-width cols

**varfmt:** string format for variable names

**valfmt:** string format for scalar values

**vecfmt:** string format for vector values

### 8.3.3 gpkit.geometric\_program

Implement the GeometricProgram class

**class** `gpkit.geometric_program.GeometricProgram` (*cost, constraints, verbosity=1*)

Bases: `gpkit.nomial_data.NomialData`

Standard mathematical representation of a GP.

**cost** [Constraint] Posynomial to minimize when solving

**constraints** [list of Posynomials] Constraints to maintain when solving (implicitly Posynomials  $\leq 1$ )

1) GeometricProgram does not accept equality constraints (e.g.  $x == 1$ );

instead use two inequality constraints (e.g.  $x \leq 1, 1/x \leq 1$ )

**verbosity** [int (optional)] If verbosity is greater than zero, warns about missing bounds on creation.

*solver\_out* and *solver\_log* are set during a solve *result* is set at the end of a solve

```
>>> gp = gpkit.geometric_program.GeometricProgram(
    # minimize
    x,
    [ # subject to
      1/x # <= 1, implicitly
    ])
>>> gp.solve()
```

**check\_solution** (*sol*, *tol*=1e-05)  
Run a series of checks to mathematically confirm *sol* solves this GP  
**sol**: **dict** solution dict, same format as return type of *solve*()  
RuntimeWarning, if any problems are found

**feasibility\_search** (*flavour*='max', *varname*=None, \**args*, \*\**kwargs*)  
Returns a new GP for the closest feasible point of the current GP.  
**flavour** [str] Specifies the objective function minimized in the search:  
    **"max"** (default) [Apply the same slack to all constraints and] minimize that slack. Described in Eqn. 10 of [Boyd2007].  
    **"product"** [Apply a unique slack to all constraints and minimize] the product of those slacks. Useful for identifying the most problematic constraints. Described in Eqn. 11 of [Boyd2007]  
**varname** [str] LaTeX name of slack variables.  
\***args**, \*\***kwargs** Passed on to GP initialization.  
[Boyd2007] : "A tutorial on geometric programming", Optim Eng 8:67-122

**solve** (*solver*=None, *verbosity*=1, \**args*, \*\**kwargs*)  
Solves a GeometricProgram and returns the solution.  
**solver** [str or function (optional)] By default uses one of the solvers found during installation. If set to "mosek", "mosek\_cli", or "cvxopt", uses that solver. If set to a function, passes that function *cs*, *A*, *p\_idx*s, and *k*.  
**verbosity** [int (optional)] If greater than 0, prints solver name and solve time.  
\***args**, \*\***kwargs** : Passed to solver constructor and solver function.

**result** [dict] A dictionary containing the translated solver result; keys below.  
    **cost** [float] The value of the objective at the solution.  
    **variables** [dict] The value of each variable at the solution.  
    **sensitivities** [dict]  
        **monomials** [array of floats] Each monomial's dual variable value at the solution.  
        **posynomials** [array of floats] Each posynomials's dual variable value at the solution.

`gpkIt.geometric_program.genA` (*exps*, *varlocs*)  
Generates *A* matrix from *exps* and *varlocs*

**exps** [list of Hashvectors] Exponents for each monomial in a GP  
    **varlocs** [dict] Locations of each variable in *exps*

**A** [sparse Cootmatrix] Exponents of the various free variables for each monomial: rows of *A* are monomials, columns of *A* are variables.

**missingbounds** [dict] Keys: variables that lack bounds. Values: which bounds are missed.

### 8.3.4 gpkIt.signomial\_program

Implement the SignomialProgram class

**class** `gpkIt.signomial_program.SignomialProgram` (*cost*, *constraints*, *verbosity*=2)  
Bases: `object`

Prepares a collection of signomials for a SP solve.

**cost** [Constraint] Signomial to minimize when solving

**constraints** [list of Signomials] Constraints to maintain when solving (implicitly Signomials  $\leq 1$ )

**verbosity** [int (optional)] Currently has no effect: SignomialPrograms don't know anything new after being created, unlike GeometricPrograms.

*gps* is set during a solve *result* is set at the end of a solve

```
>>> gp = gpkit.geometric_program.SignomialProgram(
    # minimize
    x,
    [ # subject to
      1/x - y/x, # <= 1, implicitly
      y/10 # <= 1
    ])
>>> gp.solve()
```

**localsolve** (*solver=None, verbosity=1, x0=None, rel\_tol=0.0001, iteration\_limit=50, \*args, \*\*kwargs*)

Locally solves a SignomialProgram and returns the solution.

**solver** [str or function (optional)] By default uses one of the solvers found during installation. If set to “mosek”, “mosek\_cli”, or “cvxopt”, uses that solver. If set to a function, passes that function *cs*, *A*, *p\_idx*s, and *k*.

**verbosity** [int (optional)] If greater than 0, prints solve time and number of iterations. Each GP is created and solved with verbosity one less than this, so if greater than 1, prints solver name and time for each GP.

**x0** [dict (optional)] Initial location to approximate signomials about.

**rel\_tol** [float] Iteration ends when this is greater than the distance between two consecutive solve’s objective values.

**iteration\_limit** [int] Maximum GP iterations allowed.

**\*args, \*\*kwargs** : Passed to solver function.

**result** [dict] A dictionary containing the translated solver result.

**step** (*x0=None, verbosity=1*)

### 8.3.5 gpkit.nomials

Signomial, Posynomial, Monomial, Constraint, & MonoEQConstraint classes

**class** gpkit.nomials.**Constraint** (*left, right, oper\_ge=True*)

Bases: *gpkit.nomials.Posynomial*

A constraint of the general form posynomial  $\leq$  monomial. Stored internally (*exps*, *cs*) as a single Posynomial (self  $\leq$  1). Usually initialized via operator overloading, e.g.  $cc = y^{**2} \geq 1 + x$ . Additionally stores input format (*lhs* vs *rhs*) in *self.left* and *self.right*. Form is *self.left*  $\leq$  *self.right*.

TODO: this documentation needs to address Signomial Constraints.

**class** gpkit.nomials.**MonoEQConstraint** (*m1, m2*)

Bases: *gpkit.nomials.Constraint*

A Constraint of the form Monomial == Monomial. Stored internally as a Monomial Constraint, (1 == self).

**class** gpkit.nomials.**Monomial** (*exps=None, cs=1, require\_positive=True, simplify=True, \*\*descr*)

Bases: *gpkit.nomials.Posynomial*

A Posynomial with only one term

Same as Signomial. Note: Monomial historically supported several different init formats

These will be deprecated in the future, replaced with a single `__init__` syntax, same as Signomial.

```
class gpkit.nomials.Posynomial (exps=None, cs=1, require_positive=True, simplify=True, **descr)
```

Bases: `gpkit.nomials.Signomial`

A Signomial with strictly positive cs

Same as Signomial. Note: Posynomial historically supported several different init formats

These will be deprecated in the future, replaced with a single `__init__` syntax, same as Signomial.

```
class gpkit.nomials.Signomial (exps=None, cs=1, require_positive=True, simplify=True, **descr)
```

Bases: `gpkit.nomial_data.NomialData`

A representation of a signomial.

**exps: tuple of dicts** Exponent dicts for each monomial term

**cs: tuple** Coefficient values for each monomial term

**require\_positive: bool** If True and signomials not enabled,  $c \leq 0$  will raise `ValueError`

Signomial Posynomial (if the input has only positive cs) Monomial (if the input has one term and only positive cs)

**diff** (*wrt*)

Derivative of this with respect to a Variable

**wrt (Variable):** Variable to take derivative with respect to  
Signomial (or Posynomial or Monomial)

**mono\_approximation** (*x0*)

**prod** ()

**sub** (*substitutions*, *val=None*, *require\_positive=True*)

Returns a nomial with substituted values.

`3 == (x**2 + y).sub({'x': 1, y: 2})` `3 == (x).gp.sub(x, 3)`

**substitutions** [dict or key] Either a dictionary whose keys are strings, Variables, or VarKeys, and whose values are numbers, or a string, Variable or Varkey.

**val** [number (optional)] If the substitutions entry is a single key, val holds the value

**require\_positive** [boolean (optional, default is True)] Controls whether the returned value can be a signomial.

Returns substituted nomial.

**subsummag** (*substitutions*, *val=None*)

Returns the sum of the magnitudes of the substituted Signomial.

**sum** ()

**to** (*arg*)

**value**

Self, with values substituted for variables that have values

float, if no symbolic variables remain after substitution (Monomial, Posynomial, or Signomial), otherwise.

### 8.3.6 gpkit.posyarray

Module for creating PosyArray instances.

### Example

```
>>> x = gpkit.Monomial('x')
>>> px = gpkit.PosyArray([1, x, x**2])
```

```
class gpkit.posyarray.PosyArray
```

Bases: `numpy.ndarray`

A Numpy array with elementwise inequalities and substitutions.

input\_array : array-like

```
>>> px = gpkit.PosyArray([1, x, x**2])
```

**c**

**left**

Self, sampled one index down, with zeropad

**outer** (*other*)

Returns the array and argument's outer product.

**right**

Self, sampled one index up, with zeropad

**sub** (*subs, val=None, require\_positive=True*)

Substitutes into the array

### 8.3.7 gpkit.small\_classes

Miscellaneous small classes

```
class gpkit.small_classes.CootMatrix
```

Bases: `gpkit.small_classes.CootMatrix`

A very simple sparse matrix representation.

**append** (*row, col, data*)

**dot** (*arg*)

**shape** = (`None`, `None`)

**tocoo** ()

**tocsc** ()

**tocsr** ()

Converts to a Scipy sparse `csr_matrix`

**todense** ()

**todia** ()

**todok** ()

**update\_shape** ()

```
gpkit.small_classes.CootMatrixTuple
```

alias of `CootMatrix`

```
class gpkit.small_classes.Counter
```

Bases: `object`

**class** gpkit.small\_classes.**DictOfLists**

Bases: dict

A hierarchy of dictionaries, with lists at the bottom.

**append** (*sol*)

Appends a dict (of dicts) of lists to all held lists.

**atindex** (*i*)

Indexes into each list independently.

**toarray** (*shape=None*)

Converts all lists into arrays.

**class** gpkit.small\_classes.**HashVector** (*\*args, \*\*kwargs*)

Bases: dict

A simple, sparse, string-indexed vector. Inherits from dict.

The HashVector class supports element-wise arithmetic: any undeclared variables are assumed to have a value of zero.

arg : iterable

```
>>> x = gpkit.nomials.Monomial('x')
>>> exp = gpkit.small_classes.HashVector({x: 2})
```

**class** gpkit.small\_classes.**PosyTuple** (*exps, cs, varlocs, substitutions*)

Bases: tuple

**cs**

Alias for field number 1

**exps**

Alias for field number 0

**substitutions**

Alias for field number 3

**varlocs**

Alias for field number 2

**class** gpkit.small\_classes.**SolverLog** (*verbosity=0, output=None, \*args, \*\*kwargs*)

Bases: list

Adds a *write* method to list so it's file-like and can replace stdout.

**write** (*writ*)

gpkit.small\_classes.**append\_dict** (*i, o*)

Recursively travels dict o and appends items found in i.

gpkit.small\_classes.**enlist\_dict** (*i, o*)

Recursively copies dict i into o, placing non-dict items into lists.

gpkit.small\_classes.**enray\_dict** (*i, o*)

Recursively turns lists into numpy arrays.

gpkit.small\_classes.**index\_dict** (*idx, i, o*)

Recursively travels dict i, placing items at idx into dict o.

### 8.3.8 gpkit.small\_scripts

`gpkit.small_scripts.diff(p, vk)`

`gpkit.small_scripts.flatten(ible, classes)`

Flatten an iterable that contains other iterables

**l** [Iterable] Top-level container

**out** [list] List of all objects found in the nested iterables

**TypeError** If an object is found whose class was not in classes

`gpkit.small_scripts.invalid_types_for_oper(oper, a, b)`

Raises TypeError for unsupported operations.

`gpkit.small_scripts.is_sweepvar(sub)`

Determines if a given substitution indicates a sweep.

`gpkit.small_scripts.isequal(a, b)`

`gpkit.small_scripts.latex_num(c)`

`gpkit.small_scripts.mag(c)`

Return magnitude of a Number or Quantity

`gpkit.small_scripts.mono_approx(p, x0)`

`gpkit.small_scripts.unitstr(units, into='%s', options='~', dimless='-')`

### 8.3.9 gpkit.substitution

Module containing the substitution function

`gpkit.substitution.get_constants(nomial, substitutions)`

`gpkit.substitution.separate_subs(nomial, substitutions)`

Separate substitutions by type.

`gpkit.substitution.simplify_and_mmap(signomials, subs)`

Simplifies a list of signomials and returns them with their mmaps.

**signomials** : list of Signomials

**subs** [dict] Substitutions to do before simplifying.

**signomials** [list of simplified Signomials] Signomials with cs that are solely nans and/or zeroes are removed.

**mmaps** [Map from initial monomials to substituted and simplified one.] See `small_scripts.sort_and_simplify` for more details.

`gpkit.substitution.substitution(nomial, substitutions, val=None)`

Efficient substitution into a list of monomials.

**varlocs** [dict] Dictionary mapping variables to lists of monomial indices.

**exps** [Iterable of dicts] Dictionary mapping variables to exponents, for each monomial.

**cs** [list] Coefficient for each monomial.

**substitutions** [dict] Substitutions to apply to the above.

**val** [number (optional)] Used to substitute singlet variables.

**varlocs\_** [dict] Dictionary of monomial indexes for each variable.

**exps\_** [dict] Dictionary of variable exponents for each monomial.

**cs\_** [list] Coefficients each monomial.

**subs\_** [dict] Substitutions to apply to the above.

`gpkit.substitution.vectorsub (subs, var, sub, varset)`  
Vectorized substitution

### 8.3.10 gpkit.tools module

Non-application-specific convenience methods for GPkit

`gpkit.tools.composite_objective (*objectives, **kwargs)`

`gpkit.tools.getvarkey (var)`

`gpkit.tools.getvarstr (var)`

`gpkit.tools.link (gps, varids)`

`gpkit.tools.te_exp_minus1 (posy, nterm)`  
Taylor expansion of  $e^{\text{posy}} - 1$

**posy** [gpkit.Posynomial] Variable or expression to exponentiate

**nterm** [int] Number of terms in resulting Taylor expansion

**gpkit.Posynomial** Taylor expansion of  $e^{\text{posy}} - 1$ , carried to nterm terms

`gpkit.tools.zero_lower_unbounded (model)`  
Recursively substitutes 0 for a Model's variables that lack a lower bound

### 8.3.11 gpkit.variables module

`gpkit.variables.ArrayVariable`  
alias of *VectorVariable*

`class gpkit.variables.Variable (*args, **descr)`  
Bases: *gpkit.nomials.Monomial*

**descr**

**sub** (\*args, \*\*kwargs)

Same as nomial substitution, but also allows single-argument calls

`x = Variable('x') assert x.sub(3) == Variable('x', value=3)`

**varkey**

Get the VarKey associated with this Variable

`class gpkit.variables.VectorVariable`  
Bases: *gpkit.posyarray.PosyArray*

A described vector of singlet Monomials.

**shape** [int or tuple] length or shape of resulting array



**\*args :**

may contain “name” (Strings)

“value” (Iterable) “units” (Strings + Quantity)

and/or “label” (Strings)

**\*\*descr :** VarKey description

PosyArray of Monomials, each containing a VarKey with name ‘\$name\_{i}’, where \$name is the vector’s name and i is the VarKey’s index.

### 8.3.12 gpkit.varkey module

Defines the VarKey class

**class** gpkit.varkey.**VarKey** (*name=None, \*\*kwargs*)

Bases: object

An object to correspond to each ‘variable name’.

**name** [str, VarKey, or Monomial] Name of this Variable, or object to derive this Variable from.

**\*\*kwargs :** Any additional attributes, which become the descr attribute (a dict).

VarKey with the given name and descr.

**name**

name of this VarKey

**new\_unnamed\_id** = <gpkit.small\_classes.Counter object>

**units**

units of this VarKey



---

## 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}  
}
```



---

## Acknowledgements

---

We thank the following contributors for helping to improve GPkit:

- Marshall Galbraith for setting up continuous integration.
- [Stephen Boyd](#) for inspiration and suggestions.



---

## 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)



## g

- [gpkit](#), 33
- [gpkit.geometric\\_program](#), 37
- [gpkit.model](#), 34
- [gpkit.nomials](#), 39
- [gpkit.posyarray](#), 40
- [gpkit.signomial\\_program](#), 38
- [gpkit.small\\_classes](#), 41
- [gpkit.small\\_scripts](#), 43
- [gpkit.solution\\_array](#), 36
- [gpkit.substitution](#), 43
- [gpkit.tools](#), 44
- [gpkit.variables](#), 44
- [gpkit.varkey](#), 45



## A

allsubs (gpkits.model.Model attribute), 34  
 append() (gpkits.small\_classes.CootMatrix method), 41  
 append() (gpkits.small\_classes.DictOfLists method), 42  
 append\_dict() (in module gpkits.small\_classes), 42  
 ArrayVariable (in module gpkits.variables), 44  
 atindex() (gpkits.small\_classes.DictOfLists method), 42

## C

c (gpkits.posyarray.PosyArray attribute), 41  
 check\_solution() (gpkits.geometric\_program.GeometricProgram method), 37  
 composite\_objective() (in module gpkits.tools), 44  
 constants (gpkits.model.Model attribute), 35  
 Constraint (class in gpkits.nomials), 39  
 CootMatrix (class in gpkits.small\_classes), 41  
 CootMatrixTuple (in module gpkits.small\_classes), 41  
 Counter (class in gpkits.small\_classes), 41  
 cs (gpkits.small\_classes.PosyTuple attribute), 42

## D

descr (gpkits.variables.Variable attribute), 44  
 DictOfLists (class in gpkits.small\_classes), 41  
 diff() (gpkits.nomials.Signomial method), 40  
 diff() (in module gpkits.small\_scripts), 43  
 disable\_signomials() (in module gpkits), 33  
 disable\_units() (in module gpkits), 33  
 dot() (gpkits.small\_classes.CootMatrix method), 41

## E

enable\_signomials() (in module gpkits), 34  
 enable\_units() (in module gpkits), 34  
 enlist\_dict() (in module gpkits.small\_classes), 42  
 enray\_dict() (in module gpkits.small\_classes), 42  
 exps (gpkits.small\_classes.PosyTuple attribute), 42

## F

feasibility() (gpkits.model.Model method), 35

feasibility\_search() (gpkits.geometric\_program.GeometricProgram method), 38  
 flatten() (in module gpkits.small\_scripts), 43  
 form\_program() (in module gpkits.model), 36

## G

genA() (in module gpkits.geometric\_program), 38  
 GeometricProgram (class in gpkits.geometric\_program), 37  
 get\_constants() (in module gpkits.substitution), 43  
 getvarkey() (in module gpkits.tools), 44  
 getvars() (gpkits.solution\_array.SolutionArray method), 36  
 getvarstr() (in module gpkits.tools), 44  
 gp() (gpkits.model.Model method), 35  
 gpkits (module), 33  
 gpkits.geometric\_program (module), 37  
 gpkits.model (module), 34  
 gpkits.nomials (module), 39  
 gpkits.posyarray (module), 40  
 gpkits.signomial\_program (module), 38  
 gpkits.small\_classes (module), 41  
 gpkits.small\_scripts (module), 43  
 gpkits.solution\_array (module), 36  
 gpkits.substitution (module), 43  
 gpkits.tools (module), 44  
 gpkits.variables (module), 44  
 gpkits.varkey (module), 45

## H

HashVector (class in gpkits.small\_classes), 42

## I

index\_dict() (in module gpkits.small\_classes), 42  
 invalid\_types\_for\_oper() (in module gpkits.small\_scripts), 43  
 is\_sweepvar() (in module gpkits.small\_scripts), 43  
 isequal() (in module gpkits.small\_scripts), 43

## L

latex\_num() (in module gpkIt.small\_scripts), 43  
left (gpkIt.posyarray.PosyArray attribute), 41  
link() (in module gpkIt.tools), 44  
localsolve() (gpkIt.model.Model method), 35  
localsolve() (gpkIt.signomial\_program.SignomialProgram method), 39

## M

mag() (in module gpkIt.small\_scripts), 43  
Model (class in gpkIt.model), 34  
model\_nums (gpkIt.model.Model attribute), 35  
mono\_approx() (in module gpkIt.small\_scripts), 43  
mono\_approximation() (gpkIt.nomials.Signomial method), 40  
MonoEQConstraint (class in gpkIt.nomials), 39  
Monomial (class in gpkIt.nomials), 39

## N

name (gpkIt.varkey.VarKey attribute), 45  
new\_unnamed\_id (gpkIt.varkey.VarKey attribute), 45

## O

outer() (gpkIt.posyarray.PosyArray method), 41

## P

parse\_result() (in module gpkIt.solution\_array), 37  
PosyArray (class in gpkIt.posyarray), 41  
Posynomial (class in gpkIt.nomials), 39  
PosyTuple (class in gpkIt.small\_classes), 42  
prod() (gpkIt.nomials.Signomial method), 40

## R

results\_table() (in module gpkIt.solution\_array), 37  
right (gpkIt.posyarray.PosyArray attribute), 41

## S

sens() (gpkIt.solution\_array.SolutionArray method), 36  
senssubinto() (gpkIt.solution\_array.SolutionArray method), 36  
separate\_subs() (in module gpkIt.substitution), 43  
shape (gpkIt.small\_classes.CootMatrix attribute), 41  
Signomial (class in gpkIt.nomials), 40  
SignomialProgram (class in gpkIt.signomial\_program), 38  
signomials (gpkIt.model.Model attribute), 35  
signomials\_et\_al (gpkIt.model.Model attribute), 36  
SignomialsEnabled (class in gpkIt), 33  
simplify\_and\_mmap() (in module gpkIt.substitution), 43  
SolutionArray (class in gpkIt.solution\_array), 36  
solve() (gpkIt.geometric\_program.GeometricProgram method), 38  
solve() (gpkIt.model.Model method), 36

SolverLog (class in gpkIt.small\_classes), 42  
sp() (gpkIt.model.Model method), 36  
step() (gpkIt.signomial\_program.SignomialProgram method), 39  
sub() (gpkIt.nomials.Signomial method), 40  
sub() (gpkIt.posyarray.PosyArray method), 41  
sub() (gpkIt.variables.Variable method), 44  
subinto() (gpkIt.solution\_array.SolutionArray method), 36  
substitution() (in module gpkIt.substitution), 43  
substitutions (gpkIt.small\_classes.PosyTuple attribute), 42  
subsummag() (gpkIt.nomials.Signomial method), 40  
sum() (gpkIt.nomials.Signomial method), 40

## T

table() (gpkIt.solution\_array.SolutionArray method), 37  
te\_exp\_minus1() (in module gpkIt.tools), 44  
to() (gpkIt.nomials.Signomial method), 40  
toarray() (gpkIt.small\_classes.DictOfLists method), 42  
tocoo() (gpkIt.small\_classes.CootMatrix method), 41  
tocsc() (gpkIt.small\_classes.CootMatrix method), 41  
tocsr() (gpkIt.small\_classes.CootMatrix method), 41  
todense() (gpkIt.small\_classes.CootMatrix method), 41  
todia() (gpkIt.small\_classes.CootMatrix method), 41  
todok() (gpkIt.small\_classes.CootMatrix method), 41

## U

units (gpkIt.varkey.VarKey attribute), 45  
unitstr() (in module gpkIt.small\_scripts), 43  
unsubbed (gpkIt.model.Model attribute), 36  
update\_shape() (gpkIt.small\_classes.CootMatrix method), 41

## V

value (gpkIt.nomials.Signomial attribute), 40  
Variable (class in gpkIt.variables), 44  
VarKey (class in gpkIt.varkey), 45  
varkey (gpkIt.variables.Variable attribute), 44  
varlocs (gpkIt.small\_classes.PosyTuple attribute), 42  
vectorsub() (in module gpkIt.substitution), 44  
VectorVariable (class in gpkIt.variables), 44

## W

write() (gpkIt.small\_classes.SolverLog method), 42

## Z

zero\_lower\_unbounded() (in module gpkIt.tools), 44