gpkit Documentation

Release 0.9.9

MIT Department of Aeronautics and Astronautics

Aug 20, 2020

Contents

GPkit is a Python package for defining and manipulating geometric programming (GP) models.

Our hopes are to bring the mathematics of Geometric Programming into the engineering design process in a disciplined and collaborative way, and to encourage research with and on GPs by providing an easily extensible object-oriented framework.

GPkit abstracts away the backend solver so that users can work directly with engineering equations and optimization concepts. Supported solvers are MOSEK and CVXOPT.

Join our mailing list and/or chatroom for support and examples.

CHAPTER 1

Geometric Programming 101

1.1 What is a GP?

A Geometric Program (GP) is a type of non-linear optimization problem whose objective and constraints have a particular form.

The decision variables must be strictly positive (non-zero, non-negative) quantities. This is a good fit for engineering design equations (which are often constructed to have only positive quantities), but any model with variables of unknown sign (such as forces and velocities without a predefined direction) may be difficult to express in a GP. Such models might be better expressed as *Signomials*.

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

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

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

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

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

$$g(x) = \sum_{k=1}^{K} c_k x_1^{a_1k} x_2^{a_2k} ... x_n^{a_nk}$$

For example, the expressions

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

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

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

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

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

1.2 Why are GPs special?

Geometric programs have several powerful properties:

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

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

1.3 What are Signomials / Signomial Programs?

When the coefficients in a posynomial are allowed to be negative (but the variables stay strictly positive), that is called a Signomial.

A Signomial Program has signomial constraints. While they cannot be solved as quickly or to global optima, because they build on the structure of a GP they can often be solved more quickly than a generic nonlinear program. More information can be found under *Signomial Programming*.

1.4 Where can I learn more?

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

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

CHAPTER 2

Installation

1. If you are on Mac or Windows, we recommend installing Anaconda. Alternatively, install pip and create a virtual environment.
2. (optional) Install the MOSEK solver as directed below
3. Run `pip install gpkit` in the appropriate terminal or command prompt.
4. Open a Python prompt and run `import gpkit` to finish installation and run unit tests.

If you encounter any bugs please email `gpkit@mit.edu` or raise a GitHub issue.

2.1 Installing MOSEK

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

Mac OS X

- If `which gcc` does not return anything, install the Apple Command Line Tools.
- **Download MOSEK 8, then:**
  - Move the `mosek` folder to your home directory
  - Follow these steps for Mac.
  - Request an academic license file and put it in `~/mosek/`

Linux

- **Download MOSEK 8, then:**
  - Move the `mosek` folder to your home directory
  - Follow these steps for Linux.
  - Request an academic license file and put it in `~/mosek/`

Windows

- **Download MOSEK 8, then:**
  - Follow these steps for Windows.
  - Request an academic license file and put it in `C:\Users\(your_username)\mosek\`
  - **Make sure `gcc` is on your system path.**
    - To do this, type `gcc` into a command prompt.
    - If you get `executable not found`, then install the 64-bit version (x86_64 installer architecture dropdown option) with GCC version 6.4.0 or older of mingw.
    - In an Anaconda command prompt (or equivalent), run `cd C:\Program Files\mingw-w64\x86_64-6.4.0-posix-seh-rt_v5-rev0\` (or whatever corresponds to the correct installation directory; note that if mingw is in `Program Files (x86)` instead of `Program Files` you've installed the 32-bit version by mistake)
    - Run `mingw-w64` to add it to your executable path. For step 3 of the install process you'll need to run `pip install gpkit` from this prompt.

2.2 Debugging your installation

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

- You install MOSEK after installing GPkit
- You see `Could not load settings file.` when importing GPkit, or
- `Could not load MOSEK library: ImportError('expopt.so not found.')`

To rebuild GPkit run `python -c "from gpkit.build import rebuild; rebuild()"`.

If that doesn't solve your issue then try the following:

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

2.3 Bleeding-edge installations

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

1. `pip uninstall gpkit` to uninstall your existing GPkit.
2. `git clone https://github.com/convexengineering/gpkit.git`
3. `pip install -e gpkit` to install that directory as your environment-wide GPkit.
4. `cd ..; python -c "import gpkit.tests; gpkit.tests.run()"` to test your installation from a non-local directory.

CHAPTER 3

Getting Started

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

Otherwise, *install GPkit* and import away:

```
from gpkit import Variable, VectorVariable, Model
from gpkit.nomials import Monomial, Posynomial, PosynomialInequality
```

3.1 Declaring Variables

Instances of the `Variable` class represent scalar variables. They create a `VarKey` to store the variable's name, units, a description, and value (if the Variable is to be held constant), as well as other metadata.

3.1.1 Free 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")
```

3.1.2 Fixed Variables

To declare a variable with a constant value, use the `Variable` class, as above, but put a number before the units:

```
rho = Variable("rho", 1.225, "kg/m^3", "Density of air at sea level")
```

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

```
#Declare pi equal to 3.14
pi = Variable("pi", 3.14159, "-", constant=True)
```

3.1.3 Vector Variables

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

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

3.2 Creating Monomials and Posynomials

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

```
# create a Monomial term xy^2/z
x = Variable("x")
y = Variable("y")
z = Variable("z")
m = x * y**2 / z
assert isinstance(m, Monomial)
```

```
# create a Posynomial expression x + xy^2
x = Variable("x")
y = Variable("y")
p = x + x * y**2
assert isinstance(p, Posynomial)
```

3.3 Declaring Constraints

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

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

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

3.4 Formulating a Model

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

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

```
x = Variable("x")
y = Variable("y")
z = Variable("z")
S = 200
objective = 1/(x*y*z)
constraints = [2*x*y + 2*x*z + 2*y*z <= S,
               x >= 2*y]
m = Model(objective, constraints)
```

3.5 Solving the Model

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

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

3.6 Printing Results

The solution object can represent itself as a table:

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

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

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

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

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

We can also print the optimal value and solved variables individually.

```
print("The optimal value is %s." % sol["cost"])
```

```
The optimal value is 15.5884619886.
The x dimension is 0.5774 meter.
The y dimension is 0.2887 meter.
```

3.7 Sensitivities and Dual Variables

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

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

3.7.1 Using Variable Sensitivities

Fixed variable sensitivities can be accessed from a SolutionArray's `["sensitivities"]["variables"]` dict, as in this example:

```
x = Variable("x")
x_min = Variable("x_{min}", 2)
sol = Model(x, [x_min <= x]).solve(verbosity=0)
sens_x_min = sol["sensitivities"]["variables"][x_min]
```

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:

```
x = Variable("x")
x_squared_min = Variable("x^2_{min}", 2)
sol = Model(x, [x_squared_min <= x**2]).solve(verbosity=0)
sens_x_min = sol["sensitivities"]["variables"][x_squared_min]
```

CHAPTER 4

Debugging Models

A number of errors and warnings may be raised when attempting to solve a model. A model may be primal infeasible: there is no possible solution that satisfies all constraints. A model may be dual infeasible: the optimal value of one or more variables is 0 or infinity (negative and positive infinity in logspace).

For a GP model that does not solve, solvers may be able to prove its primal or dual infeasibility, or may return an unknown status.

GPkit contains several tools for diagnosing which constraints and variables might be causing infeasibility. The first thing to do with a model `m` that won't solve is to run `m.debug()`, which will search for changes that would make the model feasible:

```
"Debug examples"
from gpkit import Variable, Model, units

x = Variable("x", "ft")
x_min = Variable("x_min", 2, "ft")
x_max = Variable("x_max", 1, "ft")
y = Variable("y", "volts")

m = Model(x/y, [x <= x_max, x >= x_min])
m.debug()

print("# Now let's try a model unsolvable with relaxed constants\n")

m2 = Model(x, [x <= units("inch"), x >= units("yard")])
m2.debug()

print("# And one that's only unbounded\n")

# the value of x_min was used up in the previous model!
x_min = Variable("x_min", 2, "ft")
m3 = Model(x/y, [x >= x_min])
m3.debug()
```

```
< DEBUGGING >
> Trying with bounded variables and relaxed constants:

Solves with these variables bounded:
 sensitive to upper bound: y
   value near upper bound: y

and these constants relaxed:
  x_min [ft]: relaxed from 2 to 1

>> Success!
# Now let's try a model unsolvable with relaxed constants

< DEBUGGING >
> Trying with bounded variables and relaxed constants:
>> Failure.
> Trying with relaxed constraints:

Solves with these constraints relaxed:
   1: 3500% relaxed, from x [ft] >= 1 [yd]
                     to 36·x [ft] >= 1 [yd]

>> Success!

# And one that's only unbounded

< DEBUGGING >
> Trying with bounded variables and relaxed constants:

Solves with these variables bounded:
 sensitive to upper bound: y
   value near upper bound: y

>> Success!
```

Note that certain modeling errors (such as omitting or forgetting a constraint) may be difficult to diagnose from this output.

4.1 Potential errors and warnings

- **`RuntimeWarning: final status of solver 'mosek' was 'DUAL_INFEAS_CER', not 'optima`**
  - The solver found a certificate of dual infeasibility: the optimal value of one or more variables is 0 or infinity. See *Dual Infeasibility* below for debugging advice.
- **`RuntimeWarning: final status of solver 'mosek' was 'PRIM_INFEAS_CER', not 'optima`**
  - The solver found a certificate of primal infeasibility: no possible solution satisfies all constraints. See *Primal Infeasibility* below for debugging advice.
- **`RuntimeWarning: final status of solver 'cvxopt' was 'unknown', not 'optimal'` or `Ru`**
  - The solver could not solve the model or find a certificate of infeasibility. This may indicate a dual infeasible model, a primal infeasible model, or other numerical issues. Try

debugging with the techniques in *Dual* and *Primal Infeasibility* below.

- **`RuntimeWarning: Primal solution violates constraint: 1.0000149786 is greater tha`**
  - this warning indicates that the solver-returned solution violates a constraint of the model, likely because the solver's tolerance for a final solution exceeds GPkit's tolerance during solution checking. This is sometimes seen in dual infeasible models, see *Dual Infeasibility* below. If you run into this, please note on this GitHub issue your solver and operating system.
- **`RuntimeWarning: Dual cost nan does not match primal cost 1.00122315152`**
  - Similarly to the above, this warning may be seen in dual infeasible models, see *Dual Infeasibility* below.

4.2 Dual Infeasibility

In some cases a model will not solve because the optimal value of one or more variables is 0 or infinity (negative or positive infinity in logspace). Such a problem is *dual infeasible* because the GP's dual problem, which determines the optimal values of the sensitivites, does not have any feasible solution. If the solver can prove that the dual is infeasible, it will return a dual infeasibility certificate. Otherwise, it may finish with a solution status of `unknown`.

`gpkit.constraints.bounded.Bounded` is a simple tool that can be used to detect unbounded variables and get dual infeasible models to solve by adding extremely large upper bounds and extremely small lower bounds to all variables in a ConstraintSet.

When a model with a Bounded ConstraintSet is solved, it checks whether any variables slid off to the bounds, notes this in the solution dictionary and prints a warning (if verbosity is greater than 0).

For example, Mosek returns `DUAL_INFEAS_CER` when attempting to solve the following model:

```python
"Demonstrate a trivial unbounded variable"
from gpkit import Variable, Model
from gpkit.constraints.bounded import Bounded

x = Variable("x")

constraints = [x >= 1]

m = Model(1/x, constraints)  # MOSEK returns DUAL_INFEAS_CER on .solve()
m = Model(1/x, Bounded(constraints))
# by default, prints bounds warning during solve
sol = m.solve(verbosity=0)
print(sol.summary())
# but they can also be accessed from the solution:
assert (sol["boundedness"]["value near upper bound"]
        == sol["boundedness"]["sensitive to upper bound"])
```

Upon viewing the printed output,

```

Solves with these variables bounded:
 sensitive to upper bound: x
   value near upper bound: x
```

(continues on next page)

(continued from previous page)

```
Optimal Cost
------------
 1e-30

Free Variables
--------------
x : 1e+30

Most Sensitive Constraints
--------------------------
    +1 : x  1e+30
```

The problem, unsurprisingly, is that the cost `1/x` has no lower bound because `x` has no upper bound.

For details read the Bounded docstring.

4.3 Primal Infeasibility

A model is primal infeasible when there is no possible solution that satisfies all constraints. A simple example is presented below.

```
"A simple primal infeasible example"
from gpkit import Variable, Model

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

m = Model(x*y, [
    x >= 1,
    y >= 2,
    x*y >= 0.5,
    x*y <= 1.5
])

# raises UnknownInfeasible on cvxopt, PrimalInfeasible on mosek
# m.solve()
```

It is not possible for `x*y` to be less than 1.5 while `x` is greater than 1 and `y` is greater than 2.

A common bug in large models that use `substitutions` is to substitute overly constraining values in for variables that make the model primal infeasible. An example of this is given below.

```
"Another simple primal infeasible example"
from gpkit import Variable, Model

x = Variable("x")
y = Variable("y", 2)

constraints = [
    x >= 1,
    0.5 <= x*y,
    x*y <= 1.5
```

(continues on next page)

(continued from previous page)

```
    ]

objective = x*y
m = Model(objective, constraints)

# raises UnknownInfeasible on cvxopt and PrimalInfeasible on mosek
# m.solve()
```

Since y is now set to 2 and x can be no less than 1, it is again impossible for x*y to be less than 1.5 and the model is primal infeasible. If y was instead set to 1, the model would be feasible and the cost would be 1.

4.3.1 Relaxation

If you suspect your model is primal infeasible, you can find the nearest primal feasible version of it by relaxing constraints: either 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.

```
"Relaxation examples"

from gpkit import Variable, Model

x = Variable("x")
x_min = Variable("x_min", 2)
x_max = Variable("x_max", 1)
m = Model(x, [x <= x_max, x >= x_min])
print("Original model")
print("==============")
print(m)
print("")
# m.solve()  # raises a RuntimeWarning!

print("With constraints relaxed equally")
print("================================")

from gpkit.constraints.relax import ConstraintsRelaxedEqually

allrelaxed = ConstraintsRelaxedEqually(m)
mr1 = Model(allrelaxed.relaxvar, allrelaxed)
print(mr1)
print(mr1.solve(verbosity=0).table())  # solves with an x of 1.414
print("")

print("With constraints relaxed individually")
print("=====================================")

from gpkit.constraints.relax import ConstraintsRelaxed

constraintsrelaxed = ConstraintsRelaxed(m)
mr2 = Model(constraintsrelaxed.relaxvars.prod() * m.cost**0.01,
            # add a bit of the original cost in
            constraintsrelaxed)
print(mr2)
```

(continues on next page)

(continued from previous page)

```
print(mr2.solve(verbosity=0).table())  # solves with an x of 1.0
print("")

print("With constants relaxed individually")
print("===================================")

from gpkit.constraints.relax import ConstantsRelaxed

constantsrelaxed = ConstantsRelaxed(m)
mr3 = Model(constantsrelaxed.relaxvars.prod() * m.cost**0.01,
            # add a bit of the original cost in
            constantsrelaxed)
print(mr3)
print(mr3.solve(verbosity=0).table())  # brings x_min down to 1.0
print("")
```

```
Original model
==============

Cost
----
 x

Constraints
-----------
 x   x_max
 x   x_min

With constraints relaxed equally
================================

Cost
----
 C

Constraints
-----------
 "minimum relaxation":
   C   1
 "relaxed constraints":
   x   C·x_max
   x_min   C·x

Optimal Cost
------------
 1.414

Free Variables
--------------
x : 1.414

  | Relax
C : 1.414

Fixed Variables
---------------
```

(continues on next page)

(continued from previous page)

```
x_max : 1
x_min : 2

Variable Sensitivities
----------------------
x_max : -0.5
x_min : +0.5

Most Sensitive Constraints
--------------------------
  +0.5 : x  C·x_max
  +0.5 : x_min  C·x


With constraints relaxed individually
=====================================

Cost
----
 C[:].prod()·x^0.01

Constraints
-----------
 "minimum relaxation":
   C[:]  1
 "relaxed constraints":
   x  C[0]·x_max
   x_min  C[1]·x

Optimal Cost
------------
 2

Free Variables
--------------
x : 1

  | Relax1
C : [ 1         2        ]

Fixed Variables
---------------
x_max : 1
x_min : 2

Variable Sensitivities
----------------------
x_min : +1
x_max : -0.99

Most Sensitive Constraints
--------------------------
    +1 : x_min  C[1]·x
 +0.99 : x  C[0]·x_max
 +0.01 : C[0]  1
```

(continues on next page)

(continued from previous page)

```
With constants relaxed individually
===================================

Cost
----
 [Relax2.x_max, Relax2.x_min].prod()·x^0.01

Constraints
-----------
 Relax2
  "original constraints":
    x  x_max
    x  x_min
  "relaxation constraints":
    "x_max":
      Relax2.x_max  1
      x_max  Relax2.OriginalValues.x_max/Relax2.x_max
      x_max  Relax2.OriginalValues.x_max·Relax2.x_max
    "x_min":
      Relax2.x_min  1
      x_min  Relax2.OriginalValues.x_min/Relax2.x_min
      x_min  Relax2.OriginalValues.x_min·Relax2.x_min

Optimal Cost
------------
 2

Free Variables
--------------
    x : 1
x_max : 1
x_min : 1

      | Relax2
x_max : 1
x_min : 2

Fixed Variables
---------------
      | Relax2.OriginalValues
x_max : 1
x_min : 2

Variable Sensitivities
----------------------
x_min : +1
x_max : -0.99

Most Sensitive Constraints
--------------------------
    +1 : x  x_min
    +1 : x_min  Relax2.OriginalValues.x_min/Relax2.x_min
 +0.99 : x  x_max
 +0.99 : x_max  Relax2.OriginalValues.x_max·Relax2.x_max
```

CHAPTER 5

Visualization and Interaction

5.1 Sensitivity Diagrams

5.1.1 Requirements

- Jupyter Notebook
- ipysankeywidget
  - Note that you'll need to activate these widgets on Jupyter by runnning
    - `jupyter nbextension enable --py --sys-prefix widgetsnbextension`
    - `jupyter nbextension enable --py --sys-prefix ipysankeywidget`

5.1.2 Example

Code in this section uses the CE solar model

```
from solar.solar import *
Vehicle = Aircraft(Npod=3, sp=True)
M = Mission(Vehicle, latitude=[20])
M.cost = M[M.aircraft.Wtotal]
sol = M.localsolve("mosek_cli")

from gpkit.interactive.sankey import Sankey
```

Once the code above has been run in a Jupyter notebook, the code below will create interactive hierarchies of your model's sensitivities, like so:

5.1.3 Explanation

Sankey diagrams can be used to visualize sensitivity structure in a model. A blue flow from a constraint to its parent indicates that the sensitivity of the chosen variable (or of making the constraint easier, if no variable is given) is negative; that is, the objective of the overall model would improve if that variable's value were increased *in that constraint alone*. Red indicates a positive sensitivity: the objective and the the constraint 'want' that variable's value decreased. Gray flows indicate a sensitivity whose absolute value is below `1e-2`, i.e. a constraint that is inactive for that variable. Where equal red and blue flows meet, they cancel each other out to gray.

5.1.4 Usage

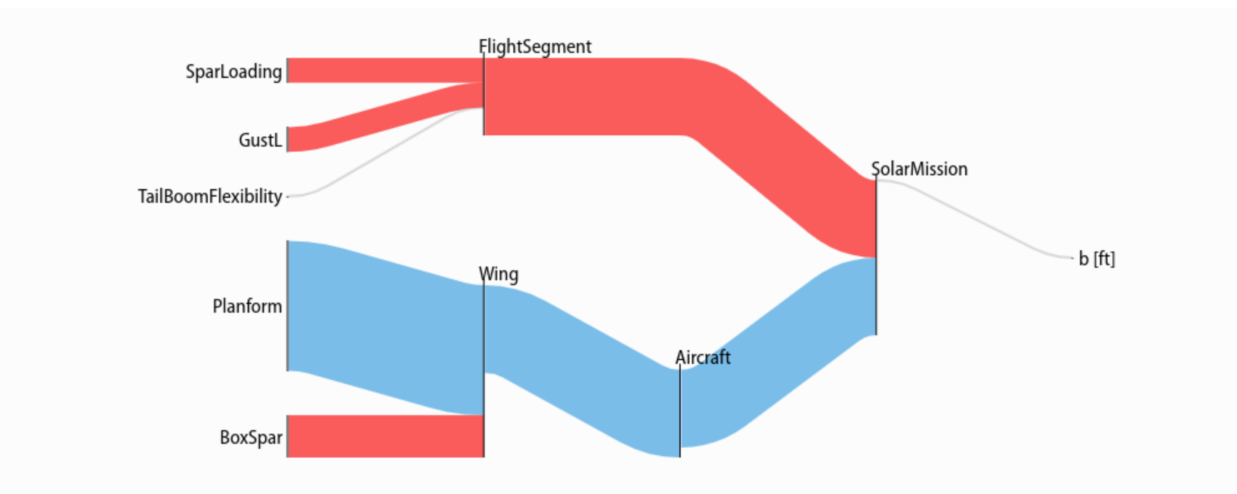
Variables

In a Sankey diagram of a variable, the variable is on the right with its final sensitivity; to the left of it are all constraints that variable is in.

Free

Free variables have an overall sensitivity of 0, so this visualization shows how the various pressures on that variable in all its constraints cancel each other out; this can get quite complex, as in this diagram of the pressures on wingspan (right-click and open in a new tab to see it more clearly):

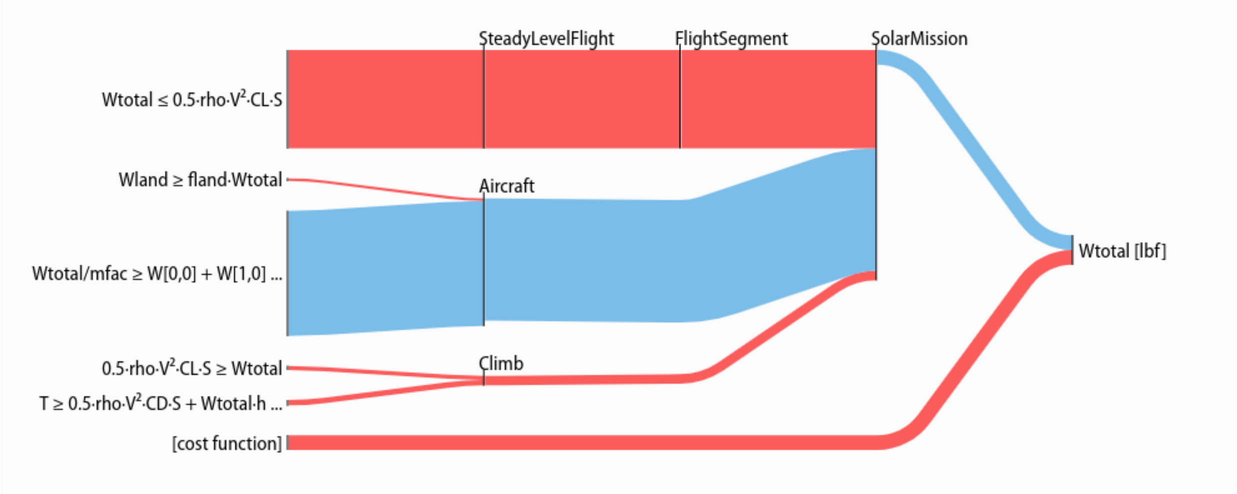
```
Sankey(sol, M, "SolarMission").diagram(M.aircraft.wing.planform.b,
↪showconstraints=False)
```



Gray lines in this diagram indicate constraints or constraint sets that the variable is in but which have no net sensitivity to it. Note that the `showconstraints` argument can be used to hide constraints if you wish to see more of the model hierarchy with the same number of links.

Variable in the cost function, have a "[cost function]" node on the diagram like so:

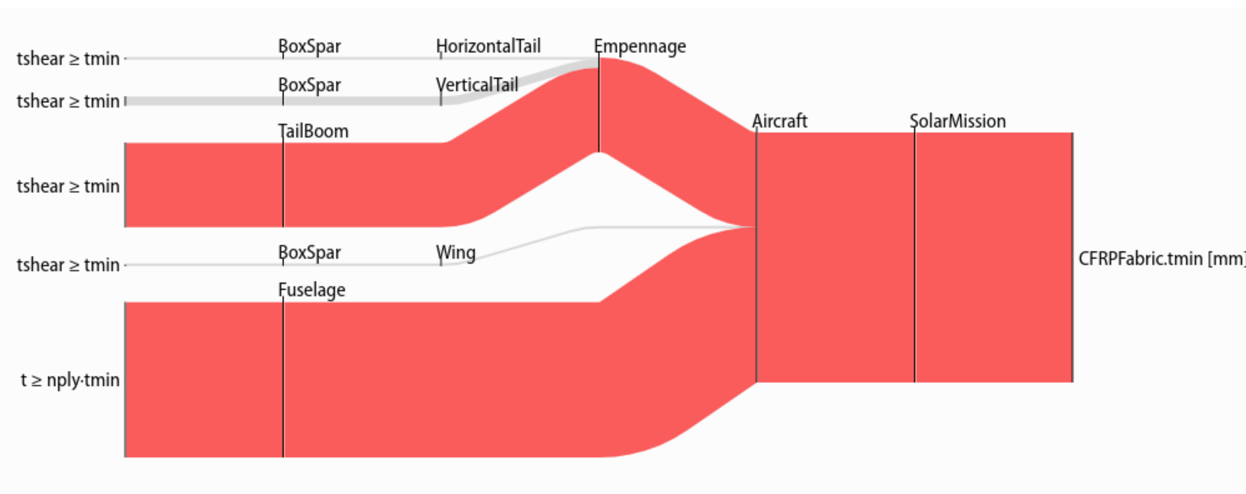
```
Sankey(sol, M, "SolarMission").diagram(M.aircraft.Wtotal)
```



Fixed

Fixed variables can have a nonzero overall sensitivity. Sankey diagrams can how that sensitivity comes together:

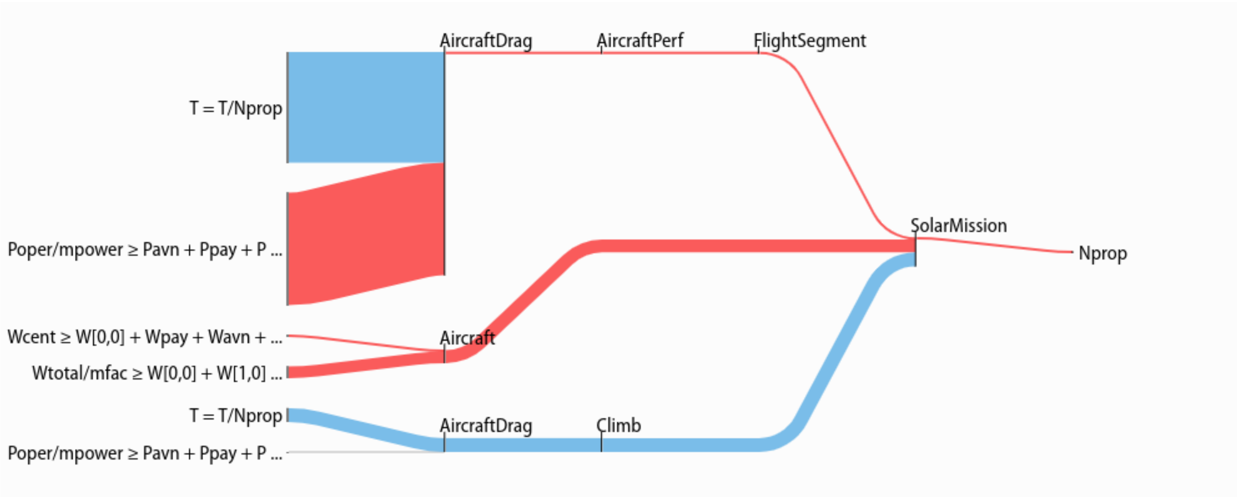
```
Sankey(sol, M, "SolarMission").diagram(M.variables_byname("tmin")[0],␣
↪left=100)
```



Note that the `left=` syntax is used to reduce the left margin in this plot. Similar arguments exist for the `right`, `top`, and `bottom` margins: all arguments are in pixels.

The only difference between free and fixed variables from this perspective is their final sensitivity; for example `Nprop`, the number of propellers on the plane, has almost zero sensitivity, much like the wingspan `b`, above.

```
Sankey(sol, M, "SolarMission").diagram(M["Nprop"])
```



Models

When created without a variable, the diagram shows the sensitivity of every named model to becoming locally easier. Because derivatives are additive, these sensitivities are too: a model's sensitivity is equal to the sum of its constraints' sensitivities and the magnitude of its fixed-variable sensitivities. Gray lines in this diagram indicate models without any tight constraints or sensitive fixed variables.

```
Sankey(sol, M, "SolarMission").diagram(maxlinks=30, showconstraints=False,␣
↪height=700)
```

Note that in addition to the `showconstraints` syntax introduced above, this uses two additional arguments you may find useful when visualizing large models: `height` sets the height of the diagram in pixels (similarly for `width`), while `maxlinks` increases the maximum number of links (default 20), making a more detailed plot. Plot construction time goes approximately as the square of the number of links, so be careful when increasing `maxlinks`!

With some different arguments, the model looks like this:

```
Sankey(sol, M).diagram(minsenss=1, maxlinks=30, left=130,␣
↪showconstraints=False)
```

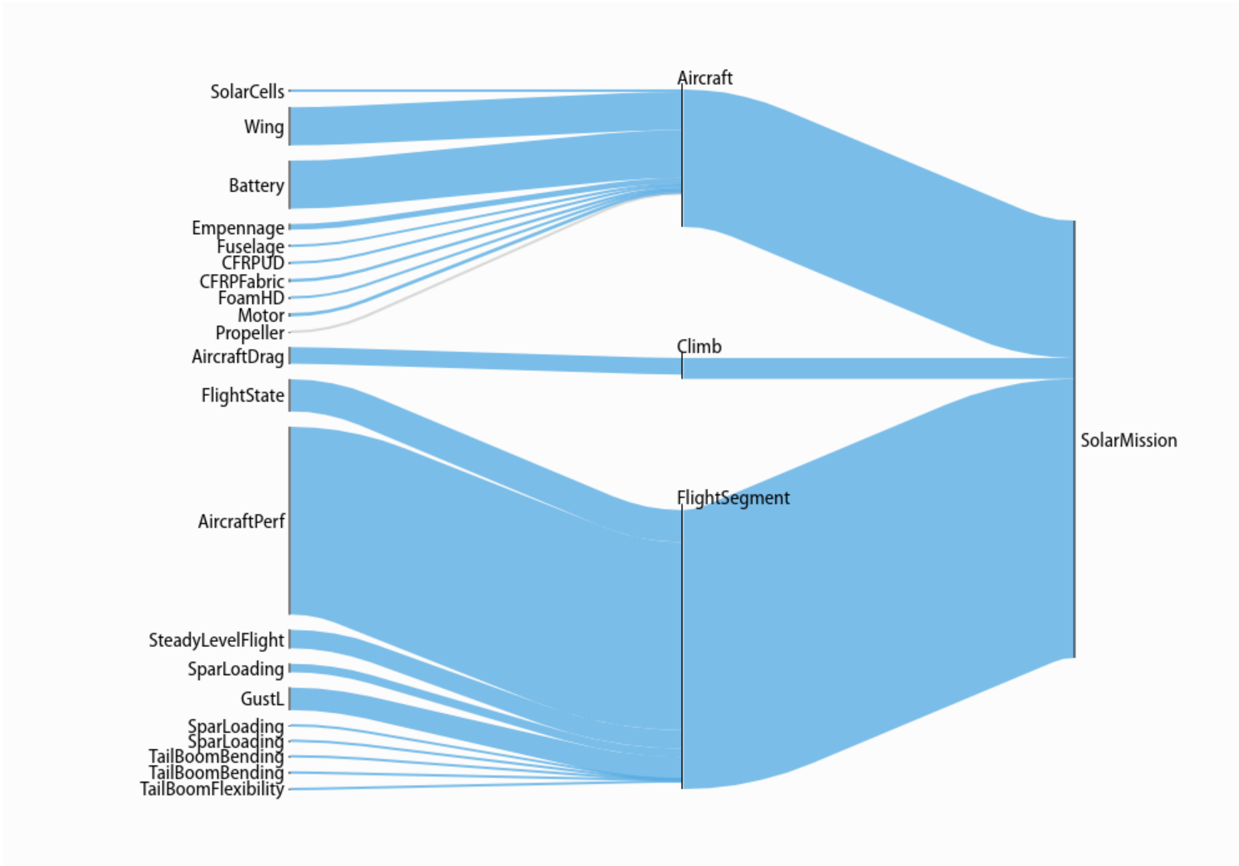
The only piece of unexplained syntax in this is `minsenss`. Perhaps unsurprisingly, this just limits the links shown to only those whose sensitivity exceeds that minimum; it's quite useful for exploring a large model.

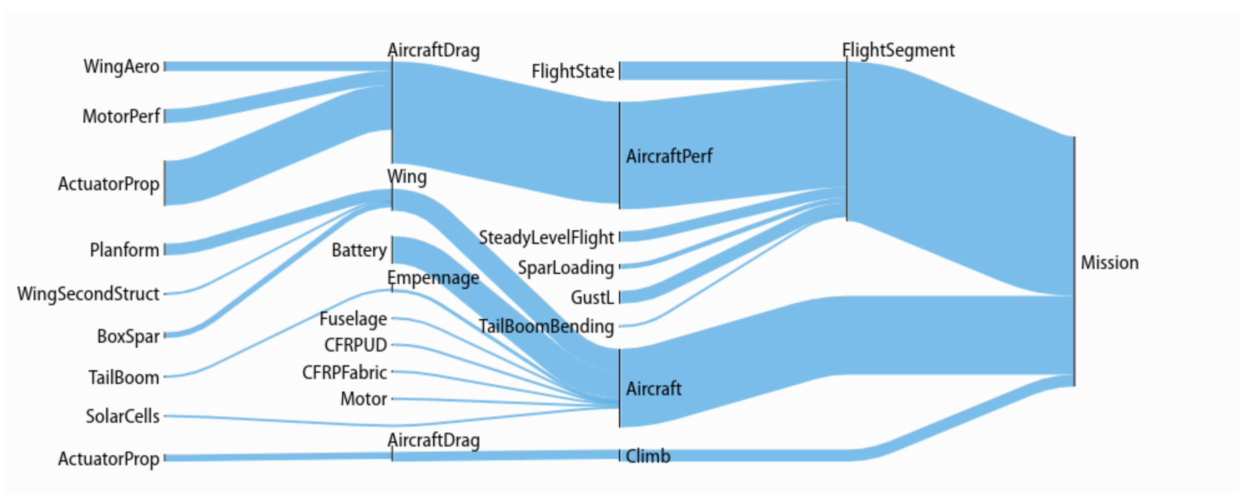
5.2 Plotting a 1D Sweep

Methods exist to facilitate creating, solving, and plotting the results of a single-variable sweep (see *Sweeps* for details). Example usage is as follows:

```
"Demonstrates manual and auto sweeping and plotting"
import matplotlib as mpl
mpl.use('Agg')
# comment out the lines above to show figures in a window
import numpy as np
from gpkit import Model, Variable, units
```

(continues on next page)

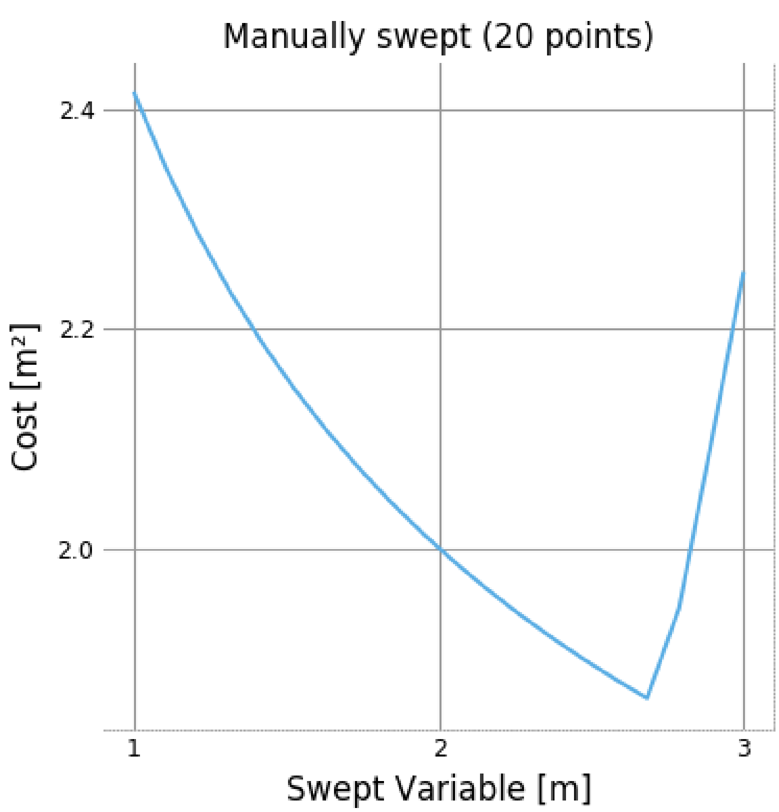

SolarCells
Wing
Battery
Empennage
Fuselage
CFRPUD
CFRPFabric
FoamHD
Motor
Propeller
AircraftDrag
FlightState
AircraftPerf
SteadyLevelFlight
SparLoading
GustL
SparLoading
SparLoading
TailBoomBending
TailBoomBending
TailBoomFlexibility
Aircraft
Climb
FlightSegment
SolarMission

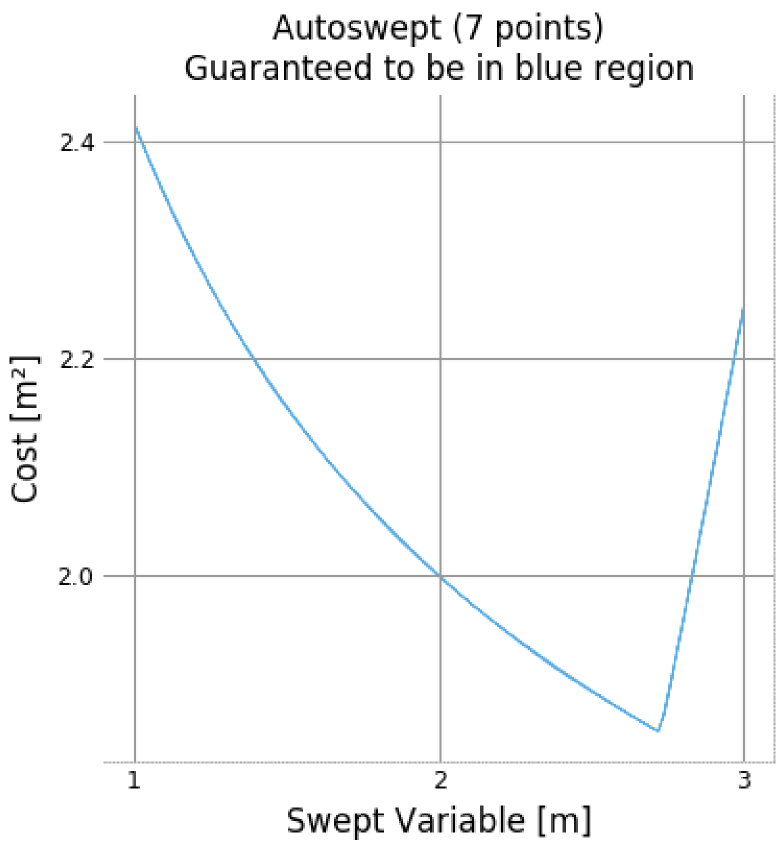

WingAero
MotorPerf
ActuatorProp
Planform
WingSecondStruct
BoxSpar
TailBoom
SolarCells
ActuatorProp
AircraftDrag
Wing
Battery
Empennage
Fuselage
CFRPUD
CFRPFabric
Motor
AircraftDrag
FlightState
AircraftPerf
SteadyLevelFlight
SparLoading
GustL
TailBoomBending
Aircraft
Climb
FlightSegment
Mission

(continued from previous page)

```
from gpkit.constraints.tight import Tight

x = Variable("x", "m", "Swept Variable")
y = Variable("y", "m^2", "Cost")
m = Model(y, [
    y >= (x/2)**-0.5 * units.m**2.5 + 1*units.m**2,
    Tight([y >= (x/2)**2])
    ])

# arguments are: model, swept: values, posnomial for y-axis
sol = m.sweep({x: np.linspace(1, 3, 20)}, verbosity=0)
f, ax = sol.plot(y)
ax.set_title("Manually swept (20 points)")
f.show()
f.savefig("plot_sweep1d.png")
sol.save()

# arguments are: model, swept: (min, max, optional logtol), posnomial for y-
↪axis
sol = m.autosweep({x: (1, 3)}, tol=0.001, verbosity=0)
f, ax = sol.plot(y)
ax.set_title("Autoswept (7 points)\nGuaranteed to be in blue region")
f.show()
f.savefig("plot_autosweep1d.png")
```

Which results in:




Autoswept (7 points)
Guaranteed to be in blue region
Cost [m²]
2.4
2.2
2.0
1
2
3
Swept Variable [m]

CHAPTER 6

Building Complex Models

6.1 Checking for result changes

Tracking the effects of changes to complex models can get out of hand; we recommend saving solutions with `sol.save()`, then checking that new solutions are almost equivalent with `sol1.almost_equal(sol2)` and/or `print(sol1.diff(sol2))`, as shown below.

```
"Example code for solution saving and differencing."
import pickle
from gpkit import Model, Variable

# build model (dummy)
# decision variable
x = Variable("x")
y = Variable("y")

# objective and constraints
objective = 0.23 + x/y # minimize x and y
constraints = [x + y <= 5, x >= 1, y >= 2]

# create model
m = Model(objective, constraints)

# solve the model
# verbosity is 0 for testing's sake, no need to do that in your code!
sol = m.solve(verbosity=0)

# save the current state of the model
sol.save("last_verified.sol")

# uncomment the line below to verify a new model
last_verified_sol = pickle.load(open("last_verified.sol", mode="rb"))
if not sol.almost_equal(last_verified_sol, reltol=1e-3):
    print(last_verified_sol.diff(sol))
```

(continues on next page)

(continued from previous page)

```
# Note you can replace the last three lines above with
# print(sol.diff("last_verified.sol"))
# if you don't mind doing the diff in that direction.
```

You can also check differences between swept solutions, or between a point solution and a sweep.

6.2 Inheriting from Model

GPkit encourages an object-oriented modeling approach, where the modeler creates objects that inherit from Model to break large systems down into subsystems and analysis domains. The benefits of this approach include modularity, reusability, and the ability to more closely follow mental models of system hierarchy. For example: two different models for a simple beam, designed by different modelers, should be able to be used interchangeably inside another subsystem (such as an aircraft wing) without either modeler having to write specifically with that use in mind.

When you create a class that inherits from Model, write a `.setup()` method to create the model's variables and return its constraints. `GPkit.Model.__init__` will call that method and automatically add your model's name and unique ID to any created variables.

Variables created in a `setup` method are added to the model even if they are not present in any constraints. This allows for simplistic 'template' models, which assume constant values for parameters and can grow incrementally in complexity as those variables are freed.

At the end of this page a detailed example shows this technique in practice.

6.3 Accessing Variables in Models

GPkit provides several ways to access a Variable in a `Model` (or `ConstraintSet`):

- using `Model.variables_byname(key)`. This returns all Variables in the Model, as well as in any submodels, that match the key.
- using `Model.__getitem__`. `Model[key]` returns the only variable matching the key, even if the match occurs in a submodel. If multiple variables match the key, an error is raised.

These methods are illustrated in the following example.

```
"Demo of accessing variables in models"
from gpkit import Model, Variable


class Battery(Model):
    """A simple battery

    Upper Unbounded
    ---------------
    m

    Lower Unbounded
    ---------------
    E
```

(continues on next page)

(continued from previous page)

```
    """
    def setup(self):
        h = Variable("h", 200, "Wh/kg", "specific energy")
        E = self.E = Variable("E", "MJ", "stored energy")
        m = self.m = Variable("m", "lb", "battery mass")
        return [E <= m*h]


class Motor(Model):
    """Electric motor

    Upper Unbounded
    ---------------
    m

    Lower Unbounded
    ---------------
    Pmax

    """
    def setup(self):
        m = self.m = Variable("m", "lb", "motor mass")
        f = Variable("f", 20, "lb/hp", "mass per unit power")
        Pmax = self.Pmax = Variable("P_{max}", "hp", "max output power")
        return [m >= f*Pmax]


class PowerSystem(Model):
    """A battery powering a motor

    Upper Unbounded
    ---------------
    m

    Lower Unbounded
    ---------------
    E, Pmax

    """
    def setup(self):
        battery, motor = Battery(), Motor()
        components = [battery, motor]
        m = self.m = Variable("m", "lb", "mass")
        self.E = battery.E
        self.Pmax = motor.Pmax

        return [components,
                m >= sum(comp.m for comp in components)]

PS = PowerSystem()
print("Getting the only var 'E': %s" % PS["E"])
print("The top-level var 'm': %s" % PS.m)
print("All the variables 'm': %s" % PS.variables_byname("m"))
```

```
Getting the only var 'E': PowerSystem.Battery.E [MJ]
The top-level var 'm': PowerSystem.m [lb]
```

(continues on next page)

(continued from previous page)

```
All the variables 'm': [gpkit.Variable(PowerSystem.Battery.m [lb]), gpkit.
↪Variable(PowerSystem.Motor.m [lb]), gpkit.Variable(PowerSystem.m [lb])]
```

6.4 Vectorization

`gpkit.Vectorize` creates an environment in which Variables are created with an additional dimension:

```
"Example Vectorize usage, from gpkit/tests/t_vars.py"
from gpkit import Variable, Vectorize, VectorVariable

with Vectorize(3):
    with Vectorize(5):
        y = Variable("y")
        x = VectorVariable(2, "x")
    z = VectorVariable(7, "z")

assert(y.shape == (5, 3))
assert(x.shape == (2, 5, 3))
assert(z.shape == (7, 3))
```

This allows models written with scalar constraints to be created with vector constraints:

```
"Vectorization demonstration"
from gpkit import Model, Variable, Vectorize

class Test(Model):
    """A simple scalar model

    Upper Unbounded
    ---------------
    x
    """
    def setup(self):
        x = self.x = Variable("x")
        return [x >= 1]

print("SCALAR")
m = Test()
m.cost = m["x"]
print(m.solve(verbosity=0).summary())

print("__________\n")
print("VECTORIZED")
with Vectorize(3):
    m = Test()
m.cost = m["x"].prod()
m.append(m["x"][1] >= 2)
print(m.solve(verbosity=0).summary())
```

```
SCALAR

Optimal Cost
------------
```

(continues on next page)

(continued from previous page)

```
 1

Free Variables
--------------
x : 1

Most Sensitive Constraints
--------------------------
    +1 : x  1

__________

VECTORIZED

Optimal Cost
------------
 2

Free Variables
--------------
x : [ 1         2         1        ]

Most Sensitive Constraints
--------------------------
    +1 : x[0]  1
    +1 : x[1]  2
    +1 : x[2]  1
```

6.5 Multipoint analysis modeling

In many engineering models, there is a physical object that is operated in multiple conditions. Some variables correspond to the design of the object (size, weight, construction) while others are vectorized over the different conditions (speed, temperature, altitude). By combining named models and vectorization we can create intuitive representations of these systems while maintaining modularity and interoperability.

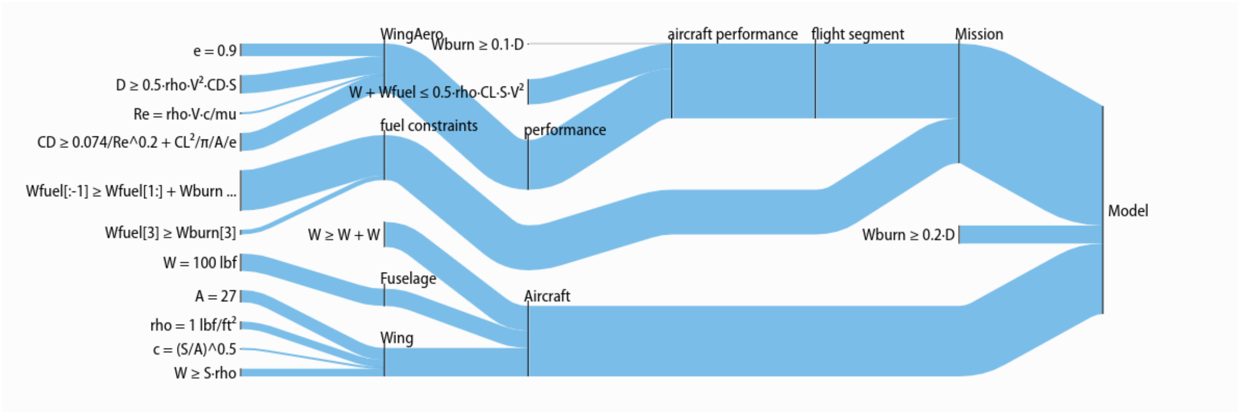
In the example below, the models `Aircraft` and `Wing` have a `.dynamic()` method which creates instances of `AircraftPerformance` and `WingAero`, respectively. The `Aircraft` and `Wing` models create variables, such as size and weight without fuel, that represent a physical object. The `dynamic` models create properties that change based on the flight conditions, such as drag and fuel weight.

This means that when an aircraft is being optimized for a mission, you can create the aircraft (`AC` in this example) and then pass it to a `Mission` model which can create vectorized aircraft performance models for each flight segment and/or flight condition.

The *sensitivity diagram* which this code outputs shows how it is organized (right-click and open in a new tab to see it more clearly):

```python
"""Modular aircraft concept"""
import pickle
import numpy as np
from gpkit import Model, Vectorize, parse_variables
```

(continues on next page)



(continued from previous page)

```
class AircraftP(Model):
    """Aircraft flight physics: weight <= lift, fuel burn

    Variables
    ---------
    Wfuel  [lbf]  fuel weight
    Wburn  [lbf]  segment fuel burn

    Upper Unbounded
    ---------------
    Wburn, aircraft.wing.c, aircraft.wing.A

    Lower Unbounded
    ---------------
    Wfuel, aircraft.W, state.mu

    """
    @parse_variables(__doc__, globals())
    def setup(self, aircraft, state):
        self.aircraft = aircraft
        self.state = state

        self.wing_aero = aircraft.wing.dynamic(aircraft.wing, state)
        self.perf_models = [self.wing_aero]

        W = aircraft.W
        S = aircraft.wing.S

        V = state.V
        rho = state.rho

        D = self.wing_aero.D
        CL = self.wing_aero.CL

        return Wburn >= 0.1*D, W + Wfuel <= 0.5*rho*CL*S*V**2, {
            "performance":
                self.perf_models}


class Aircraft(Model):
    """The vehicle model
```

(continues on next page)

(continued from previous page)

```
    Variables
    ---------
    W  [lbf]  weight

    Upper Unbounded
    ---------------
    W

    Lower Unbounded
    ---------------
    wing.c, wing.S
    """
    @parse_variables(__doc__, globals())
    def setup(self):
        self.fuse = Fuselage()
        self.wing = Wing()
        self.components = [self.fuse, self.wing]

        return [W >= sum(c.W for c in self.components),
                self.components]

    dynamic = AircraftP


class FlightState(Model):
    """Context for evaluating flight physics

    Variables
    ---------
    V     40       [knots]    true airspeed
    mu    1.628e-5 [N*s/m^2]  dynamic viscosity
    rho   0.74     [kg/m^3]   air density

    """
    @parse_variables(__doc__, globals())
    def setup(self):
        pass


class FlightSegment(Model):
    """Combines a context (flight state) and a component (the aircraft)

    Upper Unbounded
    ---------------
    Wburn, aircraft.wing.c, aircraft.wing.A

    Lower Unbounded
    ---------------
    Wfuel, aircraft.W

    """
    def setup(self, aircraft):
        self.aircraft = aircraft

        self.flightstate = FlightState()
        self.aircraftp = aircraft.dynamic(aircraft, self.flightstate)
```

(continues on next page)

(continued from previous page)

```
        self.Wburn = self.aircraftp.Wburn
        self.Wfuel = self.aircraftp.Wfuel

        return {"aircraft performance": self.aircraftp,
                "flightstate": self.flightstate}


class Mission(Model):
    """A sequence of flight segments

    Upper Unbounded
    ---------------
    aircraft.wing.c, aircraft.wing.A

    Lower Unbounded
    ---------------
    aircraft.W
    """
    def setup(self, aircraft):
        self.aircraft = aircraft

        with Vectorize(4):  # four flight segments
            self.fs = FlightSegment(aircraft)

        Wburn = self.fs.aircraftp.Wburn
        Wfuel = self.fs.aircraftp.Wfuel
        self.takeoff_fuel = Wfuel[0]

        return {
            "fuel constraints":
                [Wfuel[:-1] >= Wfuel[1:] + Wburn[:-1],
                 Wfuel[-1] >= Wburn[-1]],
            "flight segment":
                self.fs}


class WingAero(Model):
    """Wing aerodynamics

    Variables
    ---------
    CD      [-]    drag coefficient
    CL      [-]    lift coefficient
    e   0.9 [-]    Oswald efficiency
    Re      [-]    Reynold's number
    D       [lbf]  drag force

    Upper Unbounded
    ---------------
    D, Re, wing.A, state.mu

    Lower Unbounded
    ---------------
    CL, wing.S, state.mu, state.rho, state.V
    """
    @parse_variables(__doc__, globals())
```

(continues on next page)

(continued from previous page)

```
    def setup(self, wing, state):
        self.wing = wing
        self.state = state

        c = wing.c
        A = wing.A
        S = wing.S
        rho = state.rho
        V = state.V
        mu = state.mu

        return [D >= 0.5*rho*V**2*CD*S,
                Re == rho*V*c/mu,
                CD >= 0.074/Re**0.2 + CL**2/np.pi/A/e]


class Wing(Model):
    """Aircraft wing model

    Variables
    ---------
    W         [lbf]       weight
    S         [ft^2]      surface area
    rho     1 [lbf/ft^2]  areal density
    A      27 [-]         aspect ratio
    c         [ft]        mean chord

    Upper Unbounded
    ---------------
    W

    Lower Unbounded
    ---------------
    c, S
    """
    @parse_variables(__doc__, globals())
    def setup(self):
        return [c == (S/A)**0.5,
                W >= S*rho]

    dynamic = WingAero


class Fuselage(Model):
    """The thing that carries the fuel, engine, and payload

    A full model is left as an exercise for the reader.

    Variables
    ---------
    W  100 [lbf]  weight

    """
    @parse_variables(__doc__, globals())
    def setup(self):
        pass
```

(continues on next page)

(continued from previous page)

```python
AC = Aircraft()
MISSION = Mission(AC)
M = Model(MISSION.takeoff_fuel, [MISSION, AC])
print(M)
sol = M.solve(verbosity=0)
# save solution to some files
sol.savemat()
sol.savecsv()
sol.savetxt()
sol.save("solution.pkl")
# retrieve solution from a file
sol_loaded = pickle.load(open("solution.pkl", "rb"))

vars_of_interest = set(AC.varkeys)
# note that there's two ways to access submodels
assert (MISSION["flight segment"]["aircraft performance"]
        is MISSION.fs.aircraftp)
vars_of_interest.update(MISSION.fs.aircraftp.unique_varkeys)
vars_of_interest.add(M["D"])
print(sol.summary(vars_of_interest))
print(sol.table(tables=["loose constraints"]))

M.append(MISSION.fs.aircraftp.Wburn >= 0.2*MISSION.fs.aircraftp.wing_aero.D)
sol = M.solve(verbosity=0)
print(sol.diff("solution.pkl", showvars=vars_of_interest, sortbymodel=False))

# this will only make an image when run in jupyter notebook
# from gpkit.interactive.sankey import Sankey
from gpkit.interactive.sankey import Sankey
variablesankey = Sankey(sol, M).diagram(AC.wing.A)
sankey = Sankey(sol, M).diagram(width=1200, height=400, maxlinks=30)
sankey  # pylint: disable=pointless-statement
```

Note that the output table has been filtered above to show only variables of interest.

```

Cost
----
 Wfuel[0]

Constraints
-----------
 Mission
  "fuel constraints":
    Wfuel[:-1]  Wfuel[1:] + Wburn[:-1]
    Wfuel[3]  Wburn[3]

  FlightSegment
   AircraftP
    Wburn[:]  0.1·D[:]
    Aircraft.W + Wfuel[:]  0.5·rho[:]·CL[:]·S·V[:]^2
    "performance":
      WingAero
       D[:]  0.5·rho[:]·V[:]^2·CD[:]·S
       Re[:] = rho[:]·V[:]·c/mu[:]
       CD[:]  0.074/Re[:]^0.2 + CL[:]^2/π/A/e[:]
```

(continues on next page)

(continued from previous page)

```
   FlightState
    (no constraints)

 Aircraft
  Aircraft.W  Aircraft.Fuselage.W + Aircraft.Wing.W
  Fuselage
   (no constraints)

  Wing
   c = (S/A)^0.5
   Aircraft.Wing.W  S·Aircraft.Wing.rho

Optimal Cost
------------
 1.091

Free Variables
--------------
      | Aircraft
    W : 144.1                                        [lbf] weight

      | Aircraft.Wing
    S : 44.14                                        [ft²] surface area
    W : 44.14                                        [lbf] weight
    c : 1.279                                        [ft]  mean chord

      | Mission.FlightSegment.AircraftP
Wburn : [ 0.274     0.273     0.272     0.272    ]  [lbf] segment fuel burn
Wfuel : [ 1.09      0.817     0.544     0.272    ]  [lbf] fuel weight

      | Mission.FlightSegment.AircraftP.WingAero
    D : [ 2.74      2.73      2.72      2.72     ]  [lbf] drag force

Variable Sensitivities
----------------------
    | Aircraft.Fuselage
  W : +0.97  weight

    | Aircraft.Wing
  A : -0.67  aspect ratio
rho : +0.43  areal density

Next Most Sensitive Variables
-----------------------------
    | Mission.FlightSegment.AircraftP.WingAero
  e : [ -0.18     -0.18     -0.18     -0.18    ] Oswald efficiency

    | Mission.FlightSegment.FlightState
  V : [ -0.22     -0.21     -0.21     -0.21    ] true airspeed
rho : [ -0.12     -0.11     -0.11     -0.11    ] air density

Most Sensitive Constraints
--------------------------
       | Aircraft
  +1.4 : .W  .Fuselage.W + .Wing.W

       | Mission
```

(continues on next page)

(continued from previous page)

```
    +1 : Wfuel[0]  Wfuel[1] + Wburn[0]
 +0.75 : Wfuel[1]  Wfuel[2] + Wburn[1]
  +0.5 : Wfuel[2]  Wfuel[3] + Wburn[2]

       | Aircraft.Wing
 +0.43 : .W  S·.rho

Insensitive Constraints |below +1e-05|
--------------------------------------
(none)

Solution Diff (for selected variables)
======================================
(argument is the baseline solution)

Constraint Differences
**********************
@@ -31,3 +31,4 @@
   Wing
    c = (S/A)^0.5
    Aircraft.Wing.W  S·Aircraft.Wing.rho
+ Wburn[:]  0.2·D[:]

**********************

Relative Differences |above 1%|
-------------------------------
Wburn : [ +102.1%   +101.6%   +101.1%   +100.5%  ] segment fuel burn
Wfuel : [ +101.3%   +101.1%   +100.8%   +100.5%  ] fuel weight
    D : [   +1.1%       -         -         -    ] drag force
```

CHAPTER 7

Advanced Commands

7.1 Derived Variables

7.1.1 Evaluated Fixed Variables

Some fixed variables may be derived from the values of other fixed variables. For example, air density, viscosity, and temperature are functions of altitude. These can be represented by a substitution or value that is a one-argument function accepting `model.substitutions` (for details, see *Substitutions* below).

```
"Example pre-solve evaluated fixed variable"
from gpkit import Variable, Model

# code from t_GPSubs.test_calcconst in tests/t_sub.py
x = Variable("x", "hours")
t_day = Variable("t_{day}", 12, "hours")
t_night = Variable("t_{night}", lambda c: 24 - c[t_day], "hours")

# note that t_night has a function as its value
m = Model(x, [x >= t_day, x >= t_night])
sol = m.solve(verbosity=0)
assert sol["variables"][t_night] == 12

# call substitutions
m.substitutions.update({t_day: ("sweep", [8, 12, 16])})
sol = m.solve(verbosity=0)
assert (sol["variables"][t_night] == [16, 12, 8]).all()
```

These functions are automatically differentiated with the ad package to provide more accurate sensitivities. In some cases may require using functions from the `ad.admath` instead of their python or numpy equivalents; the ad documentation contains details on how to do this.

7.1.2 Evaluated Free Variables

Some free variables may be evaluated from the values of other (non-evaluated) free variables after the optimization is performed. For example, if the efficiency ν of a motor is not a GP-compatible variable, but $(1-\nu)$ is a valid GP variable, then ν can be calculated after solving. These evaluated free variables can be represented by a `Variable` with `evalfn` metadata. Note that this variable should not be used in constructing your model!

```
"Example post-solve evaluated variable"
from gpkit import Variable, Model

# code from t_constraints.test_evalfn in tests/t_sub.py
x = Variable("x")
x2 = Variable("x^2", evalfn=lambda v: v[x]**2)
m = Model(x, [x >= 2])
m.unique_varkeys = set([x2.key])
sol = m.solve(verbosity=0)
assert abs(sol(x2) - 4) <= 1e-4
```

For evaluated variables that can be used during a solution, see *Sequential Geometric Programs*.

7.2 Sweeps

Sweeps are useful for analyzing tradeoff surfaces. A sweep "value" is an Iterable of numbers, e.g. `[1, 2, 3]`. The simplest way to sweep a model is to call `model.sweep({sweepvar: sweepvalues})`, which will return a solution array but not change the model's substitutions dictionary. If multiple `sweepvars` are given, the method will run them all as independent one-dimensional sweeps and return a list of one solution per sweep. The method `model.autosweep({sweepvar: (start, end)}, tol=0.01)` behaves very similarly, except that only the bounds of the sweep need be specified and the region in betwen will be swept to a maximum possible error of tol in the log of the cost. For details see *1D Autosweeps* below.

7.2.1 Sweep Substitutions

Alternatively, or to sweep a higher-dimensional grid, Variables can swept with a substitution value takes the form `('sweep', Iterable)`, such as `('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])` will sweep `x` over three values.

Vector variables may also be substituted for: `{y: ("sweep" ,[[1, 2], [1, 2], [1, 2]])}` will sweep $y\ \forall\ y_i \in \{1, 2\}$. These sweeps cannot be specified during Variable creation.

A Model with sweep substitutions 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.

7.2.2 1D Autosweeps

If you're only sweeping over a single variable, autosweeping lets you specify a tolerance for cost error instead of a number of exact positions to solve at. GPkit will then search the sweep segment for a locally optimal number of sweeps that can guarantee a max absolute error on the log of the cost.

Accessing variable and cost values from an autosweep is slightly different, as can be seen in this example:

```
"Show autosweep_1d functionality"
import pickle
import numpy as np
import gpkit
from gpkit import units, Variable, Model
from gpkit.tools.autosweep import autosweep_1d
from gpkit.small_scripts import mag

A = Variable("A", "m**2")
l = Variable("l", "m")

m1 = Model(A**2, [A >= l**2 + units.m**2])
tol1 = 1e-3
bst1 = autosweep_1d(m1, tol1, l, [1, 10], verbosity=0)
print("Solved after %2i passes, cost logtol +/-%.3g" % (bst1.nsols, bst1.
↪tol))
# autosweep solution accessing
l_vals = np.linspace(1, 10, 10)
sol1 = bst1.sample_at(l_vals)
print("values of l: %s" % l_vals)
print("values of A: [%s] %s" %
      (" ".join("% .1f" % n for n in sol1("A").magnitude), sol1("A").units))
cost_estimate = sol1["cost"]
cost_lb, cost_ub = sol1.cost_lb(), sol1.cost_ub()
print("cost lower bound:\n%s\n" % cost_lb)
print("cost estimate:\n%s\n" % cost_estimate)
print("cost upper bound:\n%s\n" % cost_ub)
# you can evaluate arbitrary posynomials
np.testing.assert_allclose(mag(2*sol1(A)), mag(sol1(2*A)))
assert (sol1["cost"] == sol1(A**2)).all()
# the cost estimate is the logspace mean of its upper and lower bounds
np.testing.assert_allclose((np.log(mag(cost_lb)) + np.log(mag(cost_ub)))/2,
                           np.log(mag(cost_estimate)))
# save autosweep to a file and retrieve it
bst1.save("autosweep.pkl")
bst1_loaded = pickle.load(open("autosweep.pkl", "rb"))

# this problem is two intersecting lines in logspace
m2 = Model(A**2, [A >= (l/3)**2, A >= (l/3)**0.5 * units.m**1.5])
tol2 = {"mosek": 1e-12, "cvxopt": 1e-7,
        "mosek_cli": 1e-6,
        'mosek_conif': 1e-6}[gpkit.settings["default_solver"]]
# test Model method
sol2 = m2.autosweep({l: [1, 10]}, tol2, verbosity=0)
bst2 = sol2.bst
print("Solved after %2i passes, cost logtol +/-%.3g" % (bst2.nsols, bst2.
↪tol))
print("Table of solutions used in the autosweep:")
print(bst2.solarray.table())
```

If you need access to the raw solutions arrays, the smallest simplex tree containing any given point can be gotten with `min_bst = bst.min_bst(val)`, the extents of that tree with `bst.bounds` and solutions of that tree with `bst.sols`. More information is in `help(bst)`.

7.3 Tight ConstraintSets

Tight ConstraintSets will warn if any inequalities they contain are not tight (that is, the right side does not equal the left side) after solving. This is useful when you know that a constraint *should* be tight for a given model, but representing it as an equality would be non-convex.

```
"Example Tight ConstraintSet usage"
from gpkit import Variable, Model
from gpkit.constraints.tight import Tight

Tight.reltol = 1e-2  # set the global tolerance of Tight
x = Variable('x')
x_min = Variable('x_{min}', 2)
m = Model(x, [Tight([x >= 1], reltol=1e-3),  # set the specific tolerance
              x >= x_min])
m.solve(verbosity=0)  # prints warning
```

7.4 Loose ConstraintSets

Loose ConstraintSets will warn if any GP-compatible constraints they contain are not loose (that is, their sensitivity is above some threshold after solving). This is useful when you want a constraint to be inactive for a given model because it represents an important model assumption (such as a fit only valid over a particular interval).

```
"Example Loose ConstraintSet usage"
from gpkit import Variable, Model
from gpkit.constraints.loose import Loose

Loose.reltol = 1e-4  # set the global tolerance of Loose
x = Variable('x')
x_min = Variable('x_{min}', 1)
m = Model(x, [Loose([x >= 2], senstol=1e-4),  # set the specific tolerance
              x >= x_min])
m.solve(verbosity=0)  # prints warning
```

7.5 Substitutions

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

7.5.1 Substituting into Posynomials, NomialArrays, and GPs

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

```
"Example substitution; 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
```

(continues on next page)

(continued from previous page)

```
assert p.sub({x.key: 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.

7.5.2 Substituting multiple values

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

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

7.5.3 Substituting with nonnumeric values

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

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).

7.5.4 Updating ConstraintSet substitutions

ConstraintSets have a `.substitutions` KeyDict attribute which will be substituted before solving. This KeyDict accepts variable names, VarKeys, and Variable objects as keys, and can be updated (or deleted from) like a regular Python dictionary to change the substitutions that will be used at solve-time. If a ConstraintSet itself contains ConstraintSets, it and all its elements share pointers to the same substitutions dictionary object, so that updating any one of them will update all of them.

7.5.5 Fixed Variables

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

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

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

7.5.6 Freeing Fixed Variables

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

```
"Example of freeing fixed variables"
from gpkit import Variable, Model
x = Variable("x")
y = Variable("y", 3)  # fix value to 3
m = Model(x, [x >= 1 + y, y >= 1])
# verbosity is 0 for testing's sake, no need to do that in your code!
sol = m.solve(verbosity=0)  # optimal cost is 4; y appears in sol["constants
↪"]

assert abs(sol["cost"] - 4) <= 1e-4
assert y in sol["constants"]

del m.substitutions["y"]
sol = m.solve(verbosity=0)  # optimal cost is 2; y appears in Free Variables
assert abs(sol["cost"] - 2) <= 1e-4
assert y in sol["freevariables"]
```

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

CHAPTER 8

Signomial Programming

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

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

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

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

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 `.localsolve` syntax was chosen to emphasize that signomial programming returns a local optimum. For the same reason, calling `.solve` on an SP 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.

8.1 Example Usage

```
"""Adapted from t_SP in tests/t_geometric_program.py"""
from gpkit import Model, Variable, SignomialsEnabled

# Decision variables
x = Variable('x')
y = Variable('y')
```

(continues on next page)

(continued from previous page)

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

# create and solve the SP
m = Model(x, constraints)
print(m.localsolve(verbosity=0).summary())
assert abs(m.solution(x) - 0.9) < 1e-6

# full interim solutions are available
print("x values of each GP solve (note convergence)")
print(", ".join("%.5f" % sol["freevariables"][x] for sol in m.program.
↪results))
```

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.

8.2 Sequential Geometric Programs

The method of solving local GP approximations of a non-GP compatible model can be generalized, at the cost of the general smoothness and lack of a need for trust regions that SPs guarantee.

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

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

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

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


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

objective = y

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

m = Model(objective, constraints)
print(m.solve(verbosity=0).summary())
```

```
Optimal Cost
------------
 0.7854

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

Most Sensitive Constraints
--------------------------
    +1 : x  0.785
    +1 : y  x
```

Assume we have some external code which is capable of evaluating our incompatible function:

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

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

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

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


class ExternalConstraint:
    "Class for external calling"

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

    def as_gpconstr(self, x0):
        "Returns locally-approximating GP constraint"
        # Creating a default constraint for the first solve
        if self.x not in x0:
            return (self.y >= self.x)
        # Otherwise calls external code at the current position...
        x_star = x0[self.x]
        res = external_code(x_star)
        # ...and returns a linearized posy <= 1
        return (self.y >= res * self.x/x_star)
```

and replace the incompatible constraint in our GP:

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

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

objective = y

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

m = Model(objective, constraints)
print(m.localsolve(verbosity=0).summary())
```

```

Optimal Cost
------------
 0.7071

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

Most Sensitive Constraints
--------------------------
    +1 : <external_constraint.ExternalConstraint object>
    +1 : x   0.785
```

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

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

CHAPTER 9

Examples

9.1 iPython Notebook Examples

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

9.2 A Trivial GP

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

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

# Decision variable
x = Variable("x")

# Constraint
constraints = [x >= 1]

# Objective (to minimize)
objective = x

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

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

# print selected results
print("Optimal cost:  %.4g" % sol["cost"])
print("Optimal x val: %.4g" % sol["variables"][x])
```

Of course, the optimal value is 1. Output:

```
Optimal cost:  1
Optimal x val: 1
```

9.3 Maximizing the Volume of a Box

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

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

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

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

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

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

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

# Solve the Model and print the results table
print(m.solve(verbosity=0).table())
```

The output is

```

Optimal Cost
------------
 0.003674

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

Fixed Variables
```

(continues on next page)

(continued from previous page)

```
---------------
A_{floor} : 50   [m²] upper limit, floor area
 A_{wall} : 200  [m²] 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

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

Most Sensitive Constraints
--------------------------
  +1.5 : A_{wall}  2·h·w + 2·h·d
  +0.5 : alpha  h/w
```

9.4 Water Tank

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

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

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

# because its units are incorrect the line below will print a warning
bad_monomial_equality = (M == V)

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.summary())
```

The output is:

```
Infeasible monomial equality: Cannot convert from 'V [m³]' to 'M [kg]'

Optimal Cost
------------
 1.293

Free Variables
--------------
```

(continues on next page)

(continued from previous page)

```
A : 1.293                              [m²] Surface Area of the Tank
V : 0.1                                [m³] Volume of the Tank
d : [ 0.464     0.464     0.464     ]  [m]  Dimension Vector

Variable Sensitivities
----------------------
   M : +0.67  Mass of Water in the Tank
\rho : -0.67  Density of Water in the Tank

Most Sensitive Constraints
--------------------------
    +1 : A  2·(d[0]·d[1] + d[0]·d[2] + d[1]·d[2])
 +0.67 : M = V·\rho
 +0.67 : V = d[0]·d[1]·d[2]
```

9.5 Simple Wing

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

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


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

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

(continues on next page)

(continued from previous page)

```
C_f = Variable("C_f", "-", "skin friction coefficient")
W_w = Variable("W_w", "N", "wing weight")

constraints = []

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

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

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

print("SINGLE\n======")
m = Model(D, constraints)
sol = m.solve(verbosity=0)
print(sol.summary())
# save solution to a file and retrieve it
sol.save("solution.pkl")
sol.save_compressed("solution.pgz")
print(sol.diff("solution.pkl"))

print("SWEEP\n=====")
N = 2
sweeps = {V_min: ("sweep", np.linspace(20, 25, N)),
          V: ("sweep", np.linspace(45, 55, N)), }
m.substitutions.update(sweeps)
sweepsol = m.solve(verbosity=0)
print(sweepsol.summary())
sol_loaded = pickle.load(open("solution.pkl", "rb"))
assert sol_loaded.almost_equal(SolutionArray.decompress_file("solution.pgz"))
print(sweepsol.diff(sol_loaded, absdiff=True, senssdiff=True))
```

The output is:

```
SINGLE
======

Optimal Cost
------------
 303.1

Free Variables
--------------
  A : 8.46              aspect ratio
C_D : 0.02059           Drag coefficient of wing
```

(continues on next page)

(continued from previous page)

```
C_L : 0.4988              Lift coefficent of wing
C_f : 0.003599            skin friction coefficient
  D : 303.1       [N]     total drag force
 Re : 3.675e+06           Reynold's number
  S : 16.44       [m²]    total wing area
  V : 38.15       [m/s]   cruising speed
  W : 7341        [N]     total aircraft weight
W_w : 2401        [N]     wing weight

Most Sensitive Variables
------------------------
                W_0 : +1      aircraft weight excluding wing
                  e : -0.48   Oswald efficiency factor
(\frac{S}{S_{wet}}) : +0.43   wetted area ratio
                  k : +0.43   form factor
            V_{min} : -0.37   takeoff speed

Most Sensitive Constraints
--------------------------
  +1.3 : W   W_0 + W_w
    +1 : C_D   (CDA0)/S + k·C_f·(\frac{S}{S_{wet}}) + C_L²/(π·A·e)
    +1 : D   0.5·\rho·S·C_D·V²
 +0.96 : W   0.5·\rho·S·C_L·V²
 +0.43 : C_f   0.074/Re^0.2

Solution Diff
=============
(argument is the baseline solution)

** no constraint differences **

Relative Differences |above 1%|
-------------------------------
The largest is +0%.

SWEEP
=====

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

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

Free Variables
--------------
  A : [ 6.2       4.77      8.84      7.16      ]        aspect ratio
C_D : [ 0.0146    0.0123    0.0196    0.0157    ]        Drag coefficient of␣
↪wing
C_L : [ 0.296     0.198     0.463     0.31      ]        Lift coefficent of␣
↪wing
C_f : [ 0.00333   0.00314   0.00361   0.00342   ]        skin friction␣
↪coefficient
  D : [ 338       396       294       326       ]   [N]  total drag force
```

(continues on next page)

(continued from previous page)

```
 Re : [ 5.38e+06  7.24e+06  3.63e+06  4.75e+06 ]       Reynold's number
  S : [ 18.6      17.3      12.1      11.2     ]  [m²] total wing area
  W : [ 6.85e+03  6.4e+03   6.97e+03  6.44e+03 ]  [N]  total aircraft weight
W_w : [ 1.91e+03  1.46e+03  2.03e+03  1.5e+03  ]  [N]  wing weight

Most Sensitive Variables
------------------------
                W_0 : [ +0.92     +0.85     +0.95     +0.85    ] aircraft␣
↪weight excluding wing
            V_{min} : [ -0.82     -1        -0.41     -0.71    ] takeoff␣
↪speed
                  V : [ +0.59     +0.97     +0.25     +0.75    ] cruising␣
↪speed
(\frac{S}{S_{wet}}) : [ +0.56     +0.63     +0.45     +0.54    ] wetted area␣
↪ratio
                  k : [ +0.56     +0.63     +0.45     +0.54    ] form factor

Most Sensitive Constraints (in last sweep)
------------------------------------------
    +1 : C_D  (CDA0)/S + k·C_f·(\frac{S}{S_{wet}}) + C_L²/(π·A·e)
    +1 : D  0.5·\rho·S·C_D·V²
    +1 : W  W_0 + W_w
 +0.57 : W  0.5·\rho·S·C_L·V²
 +0.54 : C_f  0.074/Re^0.2

Solution Diff
=============
(argument is the baseline solution)

** no constraint differences **

Relative Differences |above 1%|
-------------------------------
     Re : [  +46.4%    +97.1%     -1.1%    +29.2%  ] Reynold's number
    C_L : [  -40.6%    -60.2%     -7.2%    -37.9%  ] Lift coefficent of wing
      V : [  +18.0%    +44.2%    +18.0%    +44.2%  ] cruising speed
    W_w : [  -20.7%    -39.3%    -15.6%    -37.4%  ] wing weight
    C_D : [  -29.0%    -40.4%     -5.0%    -23.9%  ] Drag coefficient of wing
      A : [  -26.7%    -43.6%     +4.5%    -15.3%  ] aspect ratio
      S : [  +12.8%     +5.5%    -26.5%    -32.0%  ] total wing area
      D : [  +11.5%    +30.7%     -2.9%     +7.5%  ] total drag force
V_{min} : [   -9.1%     -9.1%    +13.6%    +13.6%  ] takeoff speed
      W : [   -6.8%    -12.8%     -5.1%    -12.2%  ] total aircraft weight
    C_f : [   -7.3%    -12.7%       -       -5.0%  ] skin friction␣
↪coefficient

Absolute Differences |above 0|
------------------------------
     Re : [ +1.7e+06  +3.6e+06  -4.1e+04  +1.1e+06 ]        Reynold's number
      W : [   -5e+02  -9.4e+02  -3.8e+02    -9e+02 ]  [N]   total aircraft␣
↪weight
    W_w : [   -5e+02  -9.4e+02  -3.8e+02    -9e+02 ]  [N]   wing weight
      D : [      +35       +93      -8.8       +23 ]  [N]   total drag force
      V : [     +6.8       +17      +6.8       +17 ]  [m/s] cruising speed
      S : [     +2.1      +0.9      -4.4      -5.3 ]  [m²]  total wing area
V_{min} : [       -2        -2        +3        +3 ]  [m/s] takeoff speed
      A : [     -2.3      -3.7     +0.38      -1.3 ]        aspect ratio
```

(continues on next page)

(continued from previous page)

```
    C_L : [      -0.2       -0.3     -0.036      -0.19 ]        Lift coefficent␣
↪of wing
    C_D : [    -0.006    -0.0083     -0.001    -0.0049 ]        Drag coefficient␣
↪of wing
    C_f : [ -0.00026   -0.00046     +8e-06   -0.00018 ]        skin friction␣
↪coefficient

Sensitivity Differences |above 0.1|
-----------------------------------
                  V : [ +0.59   +0.97   +0.25   +0.75  ] cruising speed
            V_{min} : [ -0.45   -0.67     -     -0.34  ] takeoff speed
          C_{L,max} : [ -0.23   -0.34     -     -0.17  ] max CL with flaps␣
↪down
                  e : [ +0.15   +0.25     -     +0.19  ] Oswald efficiency␣
↪factor
                W_0 : [   -     -0.17     -     -0.16  ] aircraft weight␣
↪excluding wing
               \rho : [   -     +0.13     -     +0.19  ] density of air
(\frac{S}{S_{wet}}) : [ +0.13   +0.20     -     +0.11  ] wetted area ratio
                  k : [ +0.13   +0.20     -     +0.11  ] form factor
            N_{ult} : [ -0.11   -0.18     -     -0.14  ] ultimate load factor
      W_{W_{coeff1}} : [ -0.11   -0.18     -     -0.14  ] Wing Weight␣
↪Coefficent 1
               \tau : [ +0.11   +0.18     -     +0.14  ] airfoil thickness␣
↪to chord ratio
```

9.6 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 import parse_variables, Model, ureg
from gpkit.small_scripts import mag

eps = 2e-4   # has to be quite large for consistent cvxopt printouts;
             #  normally you'd set this to something more like 1e-20


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

    Variables
    ---------
    EI    [N*m^2]   Bending stiffness
    dx    [m]       Length of an element
    L   5 [m]       Overall beam length
```

(continues on next page)

(continued from previous page)

```
    Boundary Condition Variables
    ----------------------------
    V_tip     eps [N]     Tip loading
    M_tip     eps [N*m]   Tip moment
    th_base   eps [-]     Base angle
    w_base    eps [m]     Base deflection

    Node Variables of length N
    --------------------------
    q  100*np.ones(N) [N/m]    Distributed load
    V                 [N]      Internal shear
    M                 [N*m]    Internal moment
    th                [-]      Slope
    w                 [m]      Displacement

    Upper Unbounded
    ---------------
    w_tip

    """
    @parse_variables(__doc__, globals())
    def setup(self, N=4):
        # minimize tip displacement (the last w)
        self.cost = self.w_tip = w[-1]
        return {
            "definition of dx": L == (N-1)*dx,
            "boundary_conditions": [
                V[-1] >= V_tip,
                M[-1] >= M_tip,
                th[0] >= th_base,
                w[0] >= w_base
                ],
            # below: trapezoidal integration to form a piecewise-linear
            #        approximation of loading, shear, and so on
            # shear and moment increase from tip to base (left > right)
            "shear integration":
                V[:-1] >= V[1:] + 0.5*dx*(q[:-1] + q[1:]),
            "moment integration":
                M[:-1] >= M[1:] + 0.5*dx*(V[:-1] + V[1:]),
            # slope and displacement increase from base to tip (right > left)
            "theta integration":
                th[1:] >= th[:-1] + 0.5*dx*(M[1:] + M[:-1])/EI,
            "displacement integration":
                w[1:] >= w[:-1] + 0.5*dx*(th[1:] + th[:-1])
            }


b = Beam(N=6, substitutions={"L": 6, "EI": 1.1e4, "q": 110*np.ones(6)})
sol = b.solve(verbosity=0)
print(sol.summary(maxcolumns=6))
w_gp = sol("w")  # deflection along beam

L, EI, q = sol("L"), sol("EI"), sol("q")
x = np.linspace(0, mag(L), len(q))*ureg.m  # position along beam
q = q[0]  # assume uniform loading for the check below
w_exact = q/(24*EI) * x**2 * (x**2 - 4*L*x + 6*L**2)  # analytic soln
assert max(abs(w_gp - w_exact)) <= 1.1*ureg.cm
```

(continues on next page)

(continued from previous page)

```
PLOT = False
if PLOT:  # pragma: no cover
    import matplotlib.pyplot as plt
    x_exact = np.linspace(0, L, 1000)
    w_exact = q/(24*EI) * x_exact**2 * (x_exact**2 - 4*L*x_exact + 6*L**2)
    plt.plot(x, w_gp, color='red', linestyle='solid', marker='^',
             markersize=8)
    plt.plot(x_exact, w_exact, color='blue', linestyle='dashed')
    plt.xlabel('x [m]')
    plt.ylabel('Deflection [m]')
    plt.axis('equal')
    plt.legend(['GP solution', 'Analytical solution'])
    plt.show()
```

The output is:

```
Optimal Cost
------------
 1.621

Free Variables
--------------
dx : 1.2                                                                [m]    ␣
↪Length of an element
 M : [ 1.98e+03  1.27e+03  713       317       79.2      0.0002    ]   [N·m]␣
↪Internal moment
 V : [ 660       528       396       264       132       0.0002    ]   [N]    ␣
↪Internal shear
th : [ 0.0002    0.177     0.285     0.341     0.363     0.367     ]          ␣
↪Slope
 w : [ 0.0002    0.107     0.384     0.76      1.18      1.62      ]   [m]    ␣
↪Displacement

Most Sensitive Variables
------------------------
 L : +4                                                               Overall␣
↪beam length
EI : -1                                                               Bending␣
↪stiffness
 q : [ +0.0072   +0.042    +0.12     +0.23     +0.37     +0.22     ]␣
↪Distributed load

Most Sensitive Constraints
--------------------------
    +4 : L = 5·dx
    +1 : w[5]  w[4] + 0.5·dx·(th[5] + th[4])
 +0.74 : th[2]  th[1] + 0.5·dx·(M[2] + M[1])/EI
 +0.73 : w[4]  w[3] + 0.5·dx·(th[4] + th[3])
 +0.64 : M[1]  M[2] + 0.5·dx·(V[1] + V[2])
```

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

CHAPTER 10

Glossary

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

10.1 gpkit package

10.1.1 Subpackages

gpkit.constraints package

Submodules

gpkit.constraints.array module

Implements ArrayConstraint

class gpkit.constraints.array.**ArrayConstraint**(*constraints*, *left*, *oper*, *right*)

Bases: *gpkit.constraints.single_equation.SingleEquationConstraint*, list

A ConstraintSet for prettier array-constraint printing.

ArrayConstraint gets its *sub* method from ConstrainSet, and so *left* and *right* are only used for printing.

When created by NomialArray *left* and *right* are likely to be be either NomialArrays or Varkeys of VectorVariables.

lines_without(*excluded*)

Returns lines for indentation in hierarchical printing.

gpkit.constraints.bounded module

Implements Bounded

class gpkit.constraints.bounded.**Bounded**(*constraints*, ***, *verbosity=1*, *eps=1e-30*, *lower=None*, *upper=None*)

Bases: *gpkit.constraints.set.ConstraintSet*

Bounds contained variables so as to ensure dual feasibility.

constraints [iterable] constraints whose varkeys will be bounded

verbosity [int (default 1)]

how detailed of a warning to print 0: nothing 1: print warnings

eps [float (default 1e-30)] default lower bound is eps, upper bound is 1/eps

lower [float (default None)] lower bound for all varkeys, replaces eps

upper [float (default None)] upper bound for all varkeys, replaces 1/eps

check_boundaries(*result*, ***, *verbosity=0*)
Creates (and potentially prints) a dictionary of unbounded variables.

logtol_threshold = 3

process_result(*result*)
Add boundedness to the model's solution

sens_threshold = 1e-07

gpkit.constraints.bounded.**varkey_bounds**(*varkeys*, *lower*, *upper*)
Returns constraints list bounding all varkeys.

varkeys [iterable] list of varkeys to create bounds for

lower [float] lower bound for all varkeys

upper [float] upper bound for all varkeys

gpkit.constraints.costed module

Implement CostedConstraintSet

class gpkit.constraints.costed.**CostedConstraintSet**(*cost*, *constraints*, *substitutions=None*)

Bases: *gpkit.constraints.set.ConstraintSet*

A ConstraintSet with a cost

cost : gpkit.Posynomial constraints : Iterable substitutions : dict (None)

constrained_varkeys()
Return all varkeys in the cost and non-ConstraintSet constraints

lineage = None

gpkit.constraints.gp module

Implement the GeometricProgram class

class gpkit.constraints.gp.**GeometricProgram**(*cost*, *constraints*, *substitutions*, **, *checkbounds=True*)

Bases: object

Standard mathematical representation of a GP.

solver_out and *solve_log* are set during a solve *result* is set at the end of a solve if solution status is optimal

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

check_bounds(**, *err_on_missing_bounds=False*)
Checks if any variables are unbounded, through equality constraints.

check_solution(*cost*, *primal*, *nu*, *la*, *tol*, *abstol=1e-20*)
Run checks to mathematically confirm solution solves this GP

cost: float cost returned by solver

primal: list primal solution returned by solver

nu: numpy.ndarray monomial lagrange multiplier

la: numpy.ndarray posynomial lagrange multiplier

Infeasible if any problems are found

gen()
Generates nomial and solve data (A, p_idxs) from posynomials

generate_result(*solver_out*, **, *verbosity=0*, *dual_check=True*)
Generates a full SolutionArray and checks it.

model = None

nu_by_posy = None

result
Creates and caches a result from the raw solver_out

solve(*solver=None*, **, *verbosity=1*, *gen_result=True*, ***kwargs*)
Solves a GeometricProgram and returns the solution.

solver [str or function (optional)] By default uses a solver found during installation. If "mosek_conif", "mosek_cli", or "cvxopt", uses that solver. If a function, passes that function cs, A, p_idxs, and k.

verbosity [int (default 1)] If greater than 0, prints solver name and solve time.

****kwargs :** Passed to solver constructor and solver function.

result : SolutionArray

solve_log = None

solver_out = None

v_ss = None

class gpkit.constraints.gp.**MonoEqualityIndexes**
Bases: object

Class to hold MonoEqualityIndexes

gpkit.constraints.gp.**fulfill_meq_bounds**(*missingbounds*, *meq_bounds*)
Bounds variables with monomial equalities

gpkit.constraints.gp.**gen_meq_bounds**(*missingbounds*, *exps*, *meq_idxs*)
Generate conditional monomial equality bounds

gpkit.constraints.loose module

Implements Loose

class gpkit.constraints.loose.**Loose**(*constraints*, ***, *senstol=None*)
Bases: *gpkit.constraints.set.ConstraintSet*

ConstraintSet whose inequalities must result in an equality.

process_result(*result*)
Checks that all constraints are satisfied with equality

raiseerror = False

senstol = 1e-05

gpkit.constraints.model module

Implements Model

class gpkit.constraints.model.**Model**(*cost=None*, *constraints=None*, **args*, ***kwargs*)
Bases: *gpkit.constraints.costed.CostedConstraintSet*

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 [Posynomial (optional)] Defaults to *Monomial(1)*.

constraints [ConstraintSet or 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.

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

autosweep(*sweeps*, *tol=0.01*, *samplepoints=100*, ***solveargs*)
Autosweeps {var: (start, end)} pairs in sweeps to tol.

Returns swept and sampled solutions. The original simplex tree can be accessed at sol.bst

debug(*solver=None*, *verbosity=1*, ***solveargs*)
Attempts to diagnose infeasible models.

gp(*constants=None*, ***initargs*)
Return program version of self

localsolve(*solver=None*, ***, *verbosity=1*, *skipsweepfailures=False*, ***solveargs*)
Forms a mathematical program and attempts to solve it.

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

verbosity [int (default 1)] If greater than 0 prints runtime messages. Is decremented by one and then passed to programs.

skipsweepfailures [bool (default False)] If True, when a solve errors during a sweep, skip it.

**solveargs : Passed to solve() call

sol [SolutionArray] See the SolutionArray documentation for details.

ValueError if the program is invalid. RuntimeWarning if an error occurs in solving or parsing the solution.

program = None

solution = None

solve (*solver=None*, *, *verbosity=1*, *skipsweepfailures=False*, ***solveargs*)
Forms a mathematical program and attempts to solve it.

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

verbosity [int (default 1)] If greater than 0 prints runtime messages. Is decremented by one and then passed to programs.

skipsweepfailures [bool (default False)] If True, when a solve errors during a sweep, skip it.

**solveargs : Passed to solve() call

sol [SolutionArray] See the SolutionArray documentation for details.

ValueError if the program is invalid. RuntimeWarning if an error occurs in solving or parsing the solution.

sp (*constants=None*, ***initargs*)
Return program version of self

sweep (*sweeps*, ***solveargs*)
Sweeps {var: values} pairs in sweeps. Returns swept solutions.

verify_docstring ()
Verifies docstring bounds are sufficient but not excessive.

gpkit.constraints.model.**get_relaxed** (*relaxvals*, *mapped_list*, *min_return=1*)
Determines which relaxvars are considered 'relaxed'

gpkit.constraints.prog_factories module

Scripts for generating, solving and sweeping programs

gpkit.constraints.prog_factories.**evaluate_linked** (*constants*, *linked*)
Evaluates the values and gradients of linked variables.

gpkit.constraints.prog_factories.**progify** (*program*, *return_attr=None*)
Generates function that returns a program() and optionally an attribute.

program: NomialData Class to return, e.g. GeometricProgram or SequentialGeometricProgram

return_attr: string attribute to return in addition to the program

gpkit.constraints.prog_factories.**run_sweep**(*genfunction*, *self*, *solution*, *skipsweepfailures*, *constants*, *sweep*, *linked*, *solver*, *verbosity*, ***solveargs*)
Runs through a sweep.

gpkit.constraints.prog_factories.**solvify**(*genfunction*)
Returns function for making/solving/sweeping a program.

gpkit.constraints.relax module

Models for assessing primal feasibility

class gpkit.constraints.relax.**ConstantsRelaxed**(*constraints*, ***, *include_only=None*, *exclude=None*)
Bases: *gpkit.constraints.set.ConstraintSet*

Relax constants in a constraintset.

constraints [iterable] Constraints which will be relaxed (made easier).

include_only [set (optional)] variable names must be in this set to be relaxed

exclude [set (optional)] variable names in this set will never be relaxed

relaxvars [Variable] The variables controlling the relaxation. A solved value of 1 means no relaxation was necessary or optimal for a particular constant. Higher values indicate the amount by which that constant has been made easier: e.g., a value of 1.5 means it was made 50 percent easier in the final solution than in the original problem. Of course, this can also be determined by looking at the constant's new value directly.

process_result(*result*)
Transfers the constant sensitivities back to the original constants

class gpkit.constraints.relax.**ConstraintsRelaxed**(*original_constraints*)
Bases: *gpkit.constraints.set.ConstraintSet*

Relax constraints, as in Eqn. 11 of [Boyd2007].

constraints [iterable] Constraints which will be relaxed (made easier).

relaxvars [Variable] The variables controlling the relaxation. A solved value of 1 means no relaxation was necessary or optimal for a particular constraint. Higher values indicate the amount by which that constraint has been made easier: e.g., a value of 1.5 means it was made 50 percent easier in the final solution than in the original problem.

[Boyd2007] : "A tutorial on geometric programming", Optim Eng 8:67-122

class gpkit.constraints.relax.**ConstraintsRelaxedEqually**(*original_constraints*)
Bases: *gpkit.constraints.set.ConstraintSet*

Relax constraints the same amount, as in Eqn. 10 of [Boyd2007].

constraints [iterable] Constraints which will be relaxed (made easier).

relaxvar [Variable] The variable controlling the relaxation. A solved value of 1 means no relaxation. Higher values indicate the amount by which all constraints have been made easier: e.g., a value of 1.5 means all constraints were 50 percent easier in the final solution than in the original problem.

[Boyd2007] : "A tutorial on geometric programming", Optim Eng 8:67-122

gpkit.constraints.set module

Implements ConstraintSet

class gpkit.constraints.set.**ConstraintSet**(*constraints*, *substitutions=None*)
Bases: list, *gpkit.repr_conventions.ReprMixin*

Recursive container for ConstraintSets and Inequalities

as_hmapslt1(*subs*)
Yields hmaps<=1 from self.flat()

as_view()
Return a ConstraintSetView of this ConstraintSet.

constrained_varkeys()
Return all varkeys in non-ConstraintSet constraints

flat(*yield_if_hasattr=None*)
Yields contained constraints, optionally including constraintsets.

idxlookup = {}

latex(*excluded=('units',)*)
LaTeX representation of a ConstraintSet.

lines_without(*excluded*)
Lines representation of a ConstraintSet.

name_collision_varkeys()
Returns the set of contained varkeys whose names are not unique

process_result(*result*)
Does arbitrary computation / manipulation of a program's result

There's no guarantee what order different constraints will process results in, so any changes made to the program's result should be careful not to step on other constraint's toes.

- check that an inequality was tight
- add values computed from solved variables

str_without(*excluded=('unnecessary lineage', 'units')*)
String representation of a ConstraintSet.

unique_varkeys = frozenset()

variables_byname(*key*)
Get all variables with a given name

class gpkit.constraints.set.**ConstraintSetView**(*constraintset*, *index=()*)
Bases: object

Class to access particular views on a set's variables

gpkit.constraints.set.**add_meq_bounds**(*bounded*, *meq_bounded*)
Iterates through meq_bounds until convergence

`gpkit.constraints.set.badelement(cns, i, constraint, cause='')`
Identify the bad element and raise a ValueError

`gpkit.constraints.set.flatiter(iterable, yield_if_hasattr=None)`
Yields contained constraints, optionally including constraintsets.

`gpkit.constraints.set.recursively_line(iterable, excluded)`
Generates lines in a recursive tree-like fashion, the better to indent.

`gpkit.constraints.set.sort_constraints_dict(iterable)`
Sort a dictionary of {k: constraint} and return its keys and values

gpkit.constraints.sgp module

Implement the SequentialGeometricProgram class

`class gpkit.constraints.sgp.SequentialGeometricProgram(cost, model, substitutions, *, use_pccp=True, pccp_penalty=200.0, checkbounds=True)`

Bases: `object`

Prepares a collection of signomials for a SP solve.

cost [Posynomial] Objective to minimize when solving

constraints [list of Constraint or SignomialConstraint objects] Constraints to maintain when solving (implicitly Signomials <= 1)

verbosity [int (optional)] Currently has no effect: SequentialGeometricPrograms 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.SequentialGeometricProgram(
                    # minimize
                    x,
                    [   # subject to
                        1/x - y/x,  # <= 1, implicitly
                        y/10  # <= 1
                    ])
>>> gp.solve()
```

`gp(x0=None, *, cleanx0=False)`
Update self._gp for x0 and return it.

`gps = None`

`localsolve(solver=None, *, verbosity=1, x0=None, reltol=0.0001, iteration_limit=50, **solveargs)`
Locally solves a SequentialGeometricProgram 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_idxs, 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.

reltol [float] Iteration ends when this is greater than the distance between two consecutive solve's objective values.

iteration_limit [int] Maximum GP iterations allowed.

mutategp: boolean Prescribes whether to mutate the previously generated GP or to create a new GP with every solve.

****solveargs :** Passed to solver function.

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

`model = None`

`result = None`

`results`
Creates and caches results from the raw solver_outs

`slack = gpkit.Variable(SGP.PCCPslack)`

`solver_outs = None`

gpkit.constraints.sigeq module

Implements SignomialEquality

class `gpkit.constraints.sigeq.`**`SignomialEquality`**(*left*, *right*)
Bases: *`gpkit.constraints.set.ConstraintSet`*

A constraint of the general form posynomial == posynomial

gpkit.constraints.single_equation module

Implements SingleEquationConstraint

class `gpkit.constraints.single_equation.`**`SingleEquationConstraint`**(*left*, *oper*, *right*)
Bases: *`gpkit.repr_conventions.ReprMixin`*

Constraint expressible in a single equation.

`func_opers = {'<=': <built-in function le>, '=': <built-in function eq>, '>=':`

`latex`(*excluded='units'*)
Latex representation without attributes in excluded list

`latex_opers = {'<=': '\\leq', '=': '=', '>=': '\\geq'}`

`str_without`(*excluded='units'*)
String representation without attributes in excluded list

`unicode_opers = {'<=': '', '=': '=', '>=': ''}`

gpkit.constraints.tight module

Implements Tight

class gpkit.constraints.tight.**Tight**(*constraints*, *, *reltol=None*, ***kwargs*)

Bases: `gpkit.constraints.set.ConstraintSet`

ConstraintSet whose inequalities must result in an equality.

process_result(*result*)

Checks that all constraints are satisfied with equality

reltol = 0.001

Module contents

Contains ConstraintSet and related classes and objects

gpkit.interactive package

Submodules

gpkit.interactive.plot_sweep module

Implements plot_sweep1d function

gpkit.interactive.plot_sweep.**assign_axes**(*var*, *posys*, *axes*)

Assigns axes to posys, creating and formatting if necessary

gpkit.interactive.plot_sweep.**format_and_label_axes**(*var*, *posys*, *axes*, *ylabel=True*)

Formats and labels axes

gpkit.interactive.plot_sweep.**plot_1dsweepgrid**(*model*, *sweeps*, *posys*, *origsol=None*, *tol=0.01*, ***solveargs*)

Creates and plots a sweep from an existing model

Example usage: f, _ = plot_sweep_1d(m, {'x': np.linspace(1, 2, 5)}, 'y') f.savefig('mysweep.png')

gpkit.interactive.plotting module

Plotting methods

gpkit.interactive.plotting.**compare**(*models*, *sweeps*, *posys*, *tol=0.001*)

Compares the values of posys over a sweep of several models.

If posys is of the same length as models, this will plot different variables from different models.

Currently only supports a single sweepvar.

Example Usage: compare([aec, fbc], {"R": (160, 300)},

["cost", ("W_{rm batt}", "W_{rm fuel}")], tol=0.001)

gpkit.interactive.plotting.**plot_convergence**(*model*)

Plots the convergence of a signomial programming model

model: Model Signomial programming model that has already been solved

matplotlib.pyplot Figure Plot of cost as functions of SP iteration #

gpkit.interactive.sankey module

gpkit.interactive.widgets module

Module contents

Module for the interactive and plotting functions of GPkit

gpkit.nomials package

Submodules

gpkit.nomials.array module

Module for creating NomialArray instances.

Example

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

class gpkit.nomials.array.**NomialArray**

Bases: *gpkit.repr_conventions.ReprMixin*, numpy.ndarray

A Numpy array with elementwise inequalities and substitutions.

input_array : array-like

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

latex(*excluded=()*)
Returns latex representation without certain fields.

outer(*other*)
Returns the array and argument's outer product.

prod(**args, **kwargs*)
Returns a product. O(N) if no arguments and only contains monomials.

str_without(*excluded=()*)
Returns string without certain fields (such as 'lineage').

sub(*subs, require_positive=True*)
Substitutes into the array

sum(**args, **kwargs*)
Returns a sum. O(N) if no arguments are given.

vectorize(*function, *args, **kwargs*)
Apply a function to each terminal constraint, returning the array

`gpkit.nomials.array.`**`array_constraint`**(*symbol, func*)
Return function which creates constraints of the given operator.

gpkit.nomials.core module

The shared non-mathematical backbone of all Nomials

class `gpkit.nomials.core.`**`Nomial`**(*hmap*)
Bases: *gpkit.nomials.data.NomialData*

Shared non-mathematical properties of all nomials

`latex`(*excluded=()*)
Latex representation, parsing *excluded* just as .str_without does

`prod`()
Return self for compatibility with NomialArray

`str_without`(*excluded=()*)
String representation, excluding fields ('units', varkey attributes)

`sub = None`

`sum`()
Return self for compatibility with NomialArray

`value`
Self, with values substituted for variables that have values

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

`gpkit.nomials.core.`**`nomial_latex_helper`**(*c, pos_vars, neg_vars*)
Combines (varlatex, exponent) tuples, separated by positive vs negative exponent, into a single latex string.

gpkit.nomials.data module

Machinery for exps, cs, varlocs data – common to nomials and programs

class `gpkit.nomials.data.`**`NomialData`**(*hmap*)
Bases: *gpkit.repr_conventions.ReprMixin*

Object for holding cs, exps, and other basic 'nomial' properties.

cs: array (coefficient of each monomial term) exps: tuple of {VarKey: float} (exponents of each monomial term) varlocs: {VarKey: list} (terms each variable appears in) units: pint.UnitsContainer

`cs`
Create cs or return cached cs

`exps`
Create exps or return cached exps

`to`(*units*)
Create new Signomial converted to new units

`varkeys`
The NomialData's varkeys, created when necessary for a substitution.

gpkit.nomials.map module

Implements the NomialMap class

class gpkit.nomials.map.**NomialMap**

Bases: *gpkit.small_classes.HashVector*

Class for efficent algebraic represention of a nomial

A NomialMap is a mapping between hashvectors representing exponents and their coefficients in a posynomial.

For example, {{x : 1}: 2.0, {y: 1}: 3.0} represents 2*x + 3*y, where x and y are VarKey objects.

copy()

Return a copy of this

csmap = None

diff(*varkey*)

Differentiates a NomialMap with respect to a varkey

expmap = None

mmap(*orig*)

Maps substituted monomials back to the original nomial

self.expmap is the map from pre- to post-substitution exponents, and takes the form {original_exp: new_exp}

self.csmap is the map from pre-substitution exponents to coefficients.

m_from_ms is of the form {new_exp: [old_exps,]}

pmap is of the form [{orig_idx1: fraction1, orig_idx2: fraction2, },] where at the index corresponding to each new_exp is a dictionary mapping the indices corresponding to the old exps to their fraction of the post-substitution coefficient

sub(*substitutions, varkeys, parsedsubs=False*)

Applies substitutions to a NomialMap

substitutions [(dict-like)] list of substitutions to perform

varkeys [(set-like)] varkeys that are present in self (required argument so as to require efficient code)

parsedsubs [bool] flag if the substitutions have already been parsed to contain only keys in varkeys

to(*to_units*)

Returns a new NomialMap of the given units

units = None

units_of_product(*thing, thing2=None*)

Sets units to those of *thing*thing2*. Ugly optimized code.

gpkit.nomials.map.**subinplace**(*cp, exp, o_exp, vk, cval, squished*)

Modifies cp by substituting cval/expval for vk in exp

gpkit.nomials.math module

Signomial, Posynomial, Monomial, Constraint, & MonoEQCOnstraint classes

class gpkit.nomials.math.**Monomial**(*hmap=None*, *cs=1*, *require_positive=True*)
Bases: *gpkit.nomials.math.Posynomial*

A Posynomial with only one term

c
Creates c or returns a cached c

exp
Creates exp or returns a cached exp

mono_approximation(*x0*)
Monomial approximation about a point x0

x0 (dict): point to monomialize about

Monomial (unless self(x0) < 0, in which case a Signomial is returned)

class gpkit.nomials.math.**MonomialEquality**(*left*, *right*)
Bases: *gpkit.nomials.math.PosynomialInequality*

A Constraint of the form Monomial == Monomial.

as_hmapslt1(*substitutions*)
Tags posynomials for dual feasibility checking

oper = '='

sens_from_dual(*la*, *nu*, *_*)
Returns the variable/constraint sensitivities from lambda/nu

class gpkit.nomials.math.**Posynomial**(*hmap=None*, *cs=1*, *require_positive=True*)
Bases: *gpkit.nomials.math.Signomial*

A Signomial with strictly positive cs

mono_lower_bound(*x0*)
Monomial lower bound at a point x0

x0 (dict): point to make lower bound exact

Monomial

class gpkit.nomials.math.**PosynomialInequality**(*left*, *oper*, *right*)
Bases: *gpkit.nomials.math.ScalarSingleEquationConstraint*

A constraint of the general form monomial >= posynomial Stored in the posylt1_rep attribute as a single Posynomial (self <= 1) Usually initialized via operator overloading, e.g. cc = (y**2 >= 1 + x)

as_hmapslt1(*substitutions*)
Returns the posys <= 1 representation of this constraint.

feastol = 0.001

sens_from_dual(*la*, *nu*, *_*)
Returns the variable/constraint sensitivities from lambda/nu

class gpkit.nomials.math.**ScalarSingleEquationConstraint**(*left*, *oper*, *right*)
Bases: *gpkit.constraints.single_equation.SingleEquationConstraint*

A SingleEquationConstraint with scalar left and right sides.

bounded = {}

generated_by = None

meq_bounded = {}

parent = None

relaxed(*relaxvar*)
Returns the relaxation of the constraint in a list.

v_ss = None

class gpkit.nomials.math.**Signomial**(*hmap=None*, *cs=1*, *require_positive=True*)
Bases: *gpkit.nomials.core.Nomial*

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 <= 0 will raise ValueError

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

chop()
Returns a list of monomials in the signomial.

diff(*var*)
Derivative of this with respect to a Variable

var [Variable key] Variable to take derivative with respect to

Signomial (or Posynomial or Monomial)

mono_approximation(*x0*)
Monomial approximation about a point x0

x0 (dict): point to monomialize about

Monomial (unless self(x0) < 0, in which case a Signomial is returned)

posy_negy()
Get the positive and negative parts, both as Posynomials

Posynomial, Posynomial: p_pos and p_neg in (self = p_pos - p_neg) decomposition,

sub(*substitutions*, *require_positive=True*)
Returns a nomial with substitued 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.

class gpkit.nomials.math.**SignomialInequality**(*left*, *oper*, *right*)
Bases: *gpkit.nomials.math.ScalarSingleEquationConstraint*

A constraint of the general form posynomial >= posynomial

Stored at .unsubbed[0] as a single Signomial (0 >= self)

as_gpconstr (*x0*)
Returns GP-compatible approximation at x0

as_hmapslt1 (*substitutions*)
Returns the posys <= 1 representation of this constraint.

sens_from_dual (*la, nu, result*)

We want to do the following chain: dlog(Obj)/dlog(monomial[i]) = nu[i] * dlog(monomial)/d(monomial) = 1/(monomial value) * d(monomial)/d(var) = see below * d(var)/dlog(var) = var = dlog(Obj)/dlog(var)

each final monomial is really (coeff signomial)/(negy signomial)

and by the chain rule d(monomial)/d(var) = d(coeff)/d(var)*1/negy + d(1/negy)/d(var)*coeff = d(coeff)/d(var)*1/negy - d(negy)/d(var)*coeff*1/negy**2

class gpkit.nomials.math.**SingleSignomialEquality** (*left, right*)
Bases: *gpkit.nomials.math.SignomialInequality*

A constraint of the general form posynomial == posynomial

as_gpconstr (*x0*)
Returns GP-compatible approximation at x0

as_hmapslt1 (*substitutions*)
SignomialEquality is never considered GP-compatible

gpkit.nomials.substitution module

Scripts to parse and collate substitutions

gpkit.nomials.substitution.**append_sub** (*sub, keys, constants, sweep, linkedsweep*)
Appends sub to constants, sweep, or linkedsweep.

gpkit.nomials.substitution.**parse_subs** (*varkeys, substitutions, clean=False*)
Seperates subs into the constants, sweeps, linkedsweeps actually present.

gpkit.nomials.variables module

Implement Variable and ArrayVariable classes

class gpkit.nomials.variables.**ArrayVariable**
Bases: *gpkit.nomials.array.NomialArray*

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)

and/or "label" (Strings)

****descr :** VarKey description

NomialArray of Monomials, each containing a VarKey with name '$name_{i}', where $name is the vector's name and i is the VarKey's index.

class gpkit.nomials.variables.**Variable**(**args*, ***descr*)

Bases: *gpkit.nomials.math.Monomial*

A described singlet Monomial.

***args** [list]

may contain "name" (Strings)

"value" (Numbers + Quantity) or (Iterable) for a sweep "units" (Strings)

and/or "label" (Strings)

****descr** [dict] VarKey description

Monomials containing a VarKey with the name '$name', where $name is the vector's name and i is the VarKey's index.

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

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

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

to(*units*)

Create new Signomial converted to new units

class gpkit.nomials.variables.**VectorizableVariable**(**args*, ***descr*)

Bases: *gpkit.nomials.variables.Variable*, *gpkit.nomials.variables.ArrayVariable*

A Variable outside a vectorized environment, an ArrayVariable within.

gpkit.nomials.variables.**addmodelstodescr**(*descr*, *addtonamedvars=None*)

Add models to descr, optionally adding the second argument to NAMEDVARS

gpkit.nomials.variables.**veclinkedfn**(*linkedfn*, *i*)

Generate an indexed linking function.

Module contents

Contains nomials, inequalities, and arrays

gpkit.solvers package

Submodules

gpkit.solvers.cvxopt module

gpkit.solvers.mosek_cli module

Module for using the MOSEK EXPOPT command line interface

Example

```
result = _mosek.cli_expopt.imize(cs, A, p_idxs, "gpkit_mosek")
```

gpkit.solvers.mosek_cli.**assert_equal**(*received*, *expected*)

Asserts that a file's next line is as expected.

`gpkit.solvers.mosek_cli.`**`optimize_generator`**(*path=None*, ***_*)
Constructor for the MOSEK CLI solver function.

path [str (optional)] The directory in which to put the MOSEK CLI input/output files. By default uses a system-appropriate temp directory.

`gpkit.solvers.mosek_cli.`**`read_vals`**(*fil*)
Read numeric values until a blank line occurs.

`gpkit.solvers.mosek_cli.`**`remove_read_only`**(*func*, *path*, *exc*)
If we can't remove a file/directory, change permissions and try again.

`gpkit.solvers.mosek_cli.`**`write_output_file`**(*filename*, *c*, *A*, *p_idxs*)
Writes a mosekexpopt compatible GP description to *filename*.

gpkit.solvers.mosek_conif module

Module contents

gpkit.tools package

Submodules

gpkit.tools.autosweep module

Tools for optimal fits to GP sweeps

class `gpkit.tools.autosweep.`**`BinarySweepTree`**(*bounds*, *sols*, *sweptvar*, *costposy*)
Bases: `object`

Spans a line segment. May contain two subtrees that divide the segment.

bounds [two-element list] The left and right boundaries of the segment

sols [two-element list] The left and right solutions of the segment

costs [array] The left and right logcosts of the segment

splits [None or two-element list] If not None, contains the left and right subtrees

splitval [None or float] The worst-error point, where the split will be if tolerance is too low

splitlb [None or float] The cost lower bound at splitval

splitub [None or float] The cost upper bound at splitval

`add_split`(*splitval*, *splitsol*)
Creates subtrees from bounds[0] to splitval and splitval to bounds[1]

`add_splitcost`(*splitval*, *splitlb*, *splitub*)
Adds a splitval, lower bound, and upper bound

`cost_at`(*_*, *value*, *bound=None*)
Logspace interpolates between split and costs. Guaranteed bounded.

`min_bst`(*value*)
Returns smallest bst around value.

posy_at (*posy*, *value*)
Logspace interpolates between sols to get posynomial values.

No guarantees, just like a regular sweep.

sample_at (*values*)
Creates a SolutionOracle at a given range of values

save (*filename='autosweep.p'*)
Pickles the autosweep and saves it to a file.

The saved autosweep is identical except for two things:

- the cost is made unitless
- each solution's 'program' attribute is removed

Solution can then be loaded with e.g.:

```
>>> import cPickle as pickle
>>> pickle.load(open("autosweep.p"))
```

solarray
Returns a solution array of all the solutions in an autosweep

sollist
Returns a list of all the solutions in an autosweep

class gpkit.tools.autosweep.**SolutionOracle** (*bst*, *sampled_at*)
Bases: object

Acts like a SolutionArray for autosweeps

cost_lb ()
Gets cost lower bounds from the BST and units them

cost_ub ()
Gets cost upper bounds from the BST and units them

plot (*posys=None*, *axes=None*)
Plots the sweep for each posy

gpkit.tools.autosweep.**autosweep_1d** (*model*, *logtol*, *sweepvar*, *bounds*, ***solvekwargs*)
Autosweep a model over one sweepvar

gpkit.tools.autosweep.**get_tol** (*costs*, *bounds*, *sols*, *variable*)
Gets the intersection point and corresponding bounds from two solutions.

gpkit.tools.autosweep.**recurse_splits** (*model*, *bst*, *variable*, *logtol*, *solvekwargs*, *sols*)
Recursively splits a BST until logtol is reached

gpkit.tools.docstring module

Docstring-parsing methods

gpkit.tools.docstring.**check_and_parse_flag** (*string*, *flag*, *declaration_func=None*)
Checks for instances of flag in string and parses them.

gpkit.tools.docstring.**constant_declare** (*string*, *flag*, *idx2*, *countstr*)
Turns Variable declarations into Constant ones

`gpkit.tools.docstring.`**`expected_unbounded`**(*instance*, *doc*)
Gets expected-unbounded variables from a string

class `gpkit.tools.docstring.`**`parse_variables`**(*string*, *scopevars=None*)
Bases: `object`

decorator for adding local Variables from a string.

Generally called as *@parse_variables(__doc__, globals())*.

`gpkit.tools.docstring.`**`parse_varstring`**(*string*)
Parses a string to determine what variables to create from it

`gpkit.tools.docstring.`**`variable_declaration`**(*nameval*, *units*, *label*, *line*, *errorcatch=True*)
Turns parsed output into a Variable declaration

`gpkit.tools.docstring.`**`vv_declare`**(*string*, *flag*, *idx2*, *countstr*)
Turns Variable declarations into VectorVariable ones

gpkit.tools.tools module

Non-application-specific convenience methods for GPkit

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

posy [gpkit.Posynomial] Variable or expression to exponentiate

nterm [int] Number of non-constant terms in resulting Taylor expansion

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

`gpkit.tools.tools.`**`te_secant`**(*var*, *nterm*)
Taylor expansion of secant(var).

var [gpkit.monomial] Variable or expression argument

nterm [int] Number of non-constant terms in resulting Taylor expansion

gpkit.Posynomial Taylor expansion of secant(x), carried to nterm terms

`gpkit.tools.tools.`**`te_tangent`**(*var*, *nterm*)
Taylor expansion of tangent(var).

var [gpkit.monomial] Variable or expression argument

nterm [int] Number of non-constant terms in resulting Taylor expansion

gpkit.Posynomial Taylor expansion of tangent(x), carried to nterm terms

Module contents

Contains miscellaneous tools including fmincon comparison tool

10.1.2 Submodules

10.1.3 gpkit.build module

Finds solvers, sets gpkit settings, and builds gpkit

class gpkit.build.**CVXopt**
Bases: *gpkit.build.SolverBackend*

CVXopt finder.

look()
Attempts to import cvxopt.

name = 'cvxopt'

class gpkit.build.**MosekCLI**
Bases: *gpkit.build.SolverBackend*

MOSEK command line interface finder.

look()
Looks in default install locations for a mosek before version 9.

name = 'mosek_cli'

run(*where='in the default PATH'*)
Attempts to run mskexpopt.

class gpkit.build.**MosekConif**
Bases: *gpkit.build.SolverBackend*

MOSEK exponential cone solver finder.

look()
Attempts to import a mosek supporting exponential cones.

name = 'mosek_conif'

class gpkit.build.**SolverBackend**
Bases: object

Inheritable class for finding solvers. Logs.

look = None

name = None

gpkit.build.**build**()
Builds GPkit

gpkit.build.**call**(*cmd*)
Calls subprocess. Logs.

gpkit.build.**diff**(*filename*, *diff_dict*)
Applies a simple diff to a file. Logs.

gpkit.build.**isfile**(*path*)
Returns true if there's a file at $path. Logs.

gpkit.build.**log**(**args*)
Print a line and append it to the log string.

gpkit.build.**pathjoin**(**args*)
Join paths, collating multiple arguments.

`gpkit.build.`**`replacedir`**`(`*path*`)`

Replaces directory at $path. Logs.

10.1.4 gpkit.exceptions module

GPkit-specific Exception classes

exception `gpkit.exceptions.`**`DualInfeasible`**

Bases: *`gpkit.exceptions.Infeasible`*

Raised if a model returns a certificate of dual infeasibility

exception `gpkit.exceptions.`**`Infeasible`**

Bases: `RuntimeWarning`

Raised if a model does not solve

exception `gpkit.exceptions.`**`InvalidGPConstraint`**

Bases: *`gpkit.exceptions.MathematicallyInvalid`*

Raised if a non-GP-compatible constraint is used in a GP

exception `gpkit.exceptions.`**`InvalidLicense`**

Bases: `RuntimeWarning`

Raised if a solver's license is missing, invalid, or expired.

exception `gpkit.exceptions.`**`InvalidPosynomial`**

Bases: *`gpkit.exceptions.MathematicallyInvalid`*

Raised if a Posynomial would be created with a negative coefficient

exception `gpkit.exceptions.`**`InvalidSGPConstraint`**

Bases: *`gpkit.exceptions.MathematicallyInvalid`*

Raised if a non-SGP-compatible constraint is used in an SGP

exception `gpkit.exceptions.`**`MathematicallyInvalid`**

Bases: `TypeError`

Raised whenever something violates a mathematical definition.

exception `gpkit.exceptions.`**`PrimalInfeasible`**

Bases: *`gpkit.exceptions.Infeasible`*

Raised if a model returns a certificate of primal infeasibility

exception `gpkit.exceptions.`**`UnboundedGP`**

Bases: `ValueError`

Raise if a GP is not fully bounded

exception `gpkit.exceptions.`**`UnknownInfeasible`**

Bases: *`gpkit.exceptions.Infeasible`*

Raised if a model does not solve for unknown reasons

exception `gpkit.exceptions.`**`UnnecessarySGP`**

Bases: `ValueError`

Raised if an SGP is fully GP-compatible

10.1.5 gpkit.globals module

global mutable variables

class gpkit.globals.**NamedVariables**(*name*)

Bases: object

Creates an environment in which all variables have a model name and num appended to their varkeys.

lineage = ()

modelnums = {((), 'SGP'): 1}

namedvars = {}

classmethod reset_modelnumbers()

Clear all model number counters

class gpkit.globals.**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()
```

class gpkit.globals.**SignomialsEnabledMeta**

Bases: type

Metaclass to implement falsiness for SignomialsEnabled

class gpkit.globals.**Vectorize**(*dimension_length*)

Bases: object

Creates an environment in which all variables are exended in an additional dimension.

vectorization = ()

gpkit.globals.**load_settings**(*path=None*, *trybuild=True*)

Load the settings file at SETTINGS_PATH; return settings dict

10.1.6 gpkit.keydict module

Implements KeyDict and KeySet classes

class gpkit.keydict.**KeyDict**(**args*, ***kwargs*)

Bases: *gpkit.keydict.KeyMap*, dict

KeyDicts do two things over a dict: map keys and collapse arrays.

>>>> kd = gpkit.keydict.KeyDict()

For mapping keys, see KeyMapper.__doc__

If .collapse_arrays is True then VarKeys which have a *shape* parameter (indicating they are part of an array) are stored as numpy arrays, and automatically de-indexed when a matching VarKey with a particular *idx* parameter is used as a key.

See also: gpkit/tests/t_keydict.py.

collapse_arrays = True

get (*key*, **alternative*)
Return the value for key if key is in the dictionary, else default.

update (**args*, ***kwargs*)
Iterates through the dictionary created by args and kwargs

class gpkit.keydict.**KeyMap** (**args*, ***kwargs*)
Bases: `object`

Helper class to provide KeyMapping to interfaces.

A KeyMap keeps an internal list of VarKeys as canonical keys, and their values can be accessed with any object whose *key* attribute matches one of those VarKeys, or with strings matching any of the multiple possible string interpretations of each key:

For example, after creating the KeyDict kd and setting kd[x] = v (where x is a Variable or VarKey), v can be accessed with by the following keys:

- x
- x.key
- x.name (a string)
- “x_modelname” (x’s name including modelname)

Note that if a item is set using a key that does not have a *.key* attribute, that key can be set and accessed normally.

collapse_arrays = False

keymap = []

log_gets = False

parse_and_index (*key*)
Returns key if key had one, and veckey/idx for indexed veckeys.

update_keymap ()
Updates the keymap with the keys in _unmapped_keys

varkeys = None

class gpkit.keydict.**KeySet** (**args*, ***kwargs*)
Bases: *gpkit.keydict.KeyMap*, `set`

KeyMaps that don’t collapse arrays or store values.

collapse_arrays = False

update (*keys*)
Iterates through the dictionary created by args and kwargs

gpkit.keydict.**clean_value** (*key*, *value*)
Gets the value of variable-less monomials, so that *x.sub({x: gpkit.units.m})* and *x.sub({x: gpkit.ureg.m})* are equivalent.

Also converts any quantities to the key’s units, because quantities can’t/shouldn’t be stored as elements of numpy arrays.

10.1.7 gpkit.repr_conventions module

Repository for representation standards

class gpkit.repr_conventions.**ReprMixin**
Bases: object

This class combines various printing methods for easier adoption.

ast = None

cached_strs = None

latex_unitstr()
Returns latex unitstr

lineagestr(*modelnums=True*)
Returns properly formatted lineage string

parse_ast(*excluded='units'*)
Turns the AST of this object's construction into a faithful string

unitstr(*into='%s'*, *options=':P~'*, *dimless="*)
Returns the string corresponding to an object's units.

gpkit.repr_conventions.**latex_unitstr**(*units*)
Returns latex unitstr

gpkit.repr_conventions.**lineagestr**(*lineage*, *modelnums=True*)
Returns properly formatted lineage string

gpkit.repr_conventions.**parenthesize**(*string*, *addi=True*, *mult=True*)
Parenthesizes a string if it needs it and isn't already.

gpkit.repr_conventions.**strify**(*val*, *excluded*)
Turns a value into as pretty a string as possible.

gpkit.repr_conventions.**unitstr**(*units*, *into='%s'*, *options=':P~'*, *dimless="*)
Returns the string corresponding to an object's units.

10.1.8 gpkit.small_classes module

Miscellaneous small classes

class gpkit.small_classes.**CootMatrix**(*row*, *col*, *data*)
Bases: object

A very simple sparse matrix representation.

dot(*arg*)
Returns dot product with arg.

tocoo()
Converts to another type of matrix.

tocsc()
Converts to another type of matrix.

tocsr()
Converts to a Scipy sparse csr_matrix

todense()
Converts to another type of matrix.

todia()
Converts to another type of matrix.

todok()
Converts to another type of matrix.

class gpkit.small_classes.**Count**
Bases: `object`

Like python 2's itertools.count, for Python 3 compatibility.

next()
Increment self.count and return it

class gpkit.small_classes.**DictOfLists**
Bases: `dict`

A hierarchy of dicionaries, 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.

to_arrays()
Converts all lists into array.

class gpkit.small_classes.**FixedScalar**
Bases: `object`

Instances of this class are scalar Nomials with no variables

class gpkit.small_classes.**FixedScalarMeta**
Bases: `type`

Metaclass to implement instance checking for fixed scalars

class gpkit.small_classes.**HashVector**
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})
```

copy()
Return a copy of this

hashvalue = None

class gpkit.small_classes.**SolverLog**(*output=None*, *, *verbosity=0*)
Bases: `list`

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

write(*writ*)
Append and potentially write the new line.

`gpkit.small_classes.`**`matrix_converter`**(*name*)
Generates conversion function.

10.1.9 gpkit.small_scripts module

Assorted helper methods

`gpkit.small_scripts.`**`appendsolwarning`**(*msg*, *data*, *result*, *category='uncategorized'*)
Append a particular category of warnings to a solution.

`gpkit.small_scripts.`**`is_sweepvar`**(*sub*)
Determines if a given substitution indicates a sweep.

`gpkit.small_scripts.`**`mag`**(*c*)
Return magnitude of a Number or Quantity

`gpkit.small_scripts.`**`maybe_flatten`**(*value*)
Extract values from 0-d numpy arrays, if necessary

`gpkit.small_scripts.`**`splitsweep`**(*sub*)
Splits a substitution into (is_sweepvar, sweepval)

`gpkit.small_scripts.`**`try_str_without`**(*item*, *excluded*, *, *latex=False*)
Try to call item.str_without(excluded); fall back to str(item)

10.1.10 gpkit.solution_array module

Defines SolutionArray class

class `gpkit.solution_array.`**`SolSavingEnvironment`**(*solarray*)
Bases: `object`

Temporarily removes construction/solve attributes from constraints.

This approximately halves the size of the pickled solution.

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 [NomialArray] 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
```

(continues on next page)

(continued from previous page)

```
>>> senss = [sol.sens(x_min), sol.sens(x_min)]
>>> senss.append(sol["sensitivities"]["variables"]["x_{min}"])
>>> assert all(np.array(senss) == 1)
```

almost_equal (*other*, *reltol=0.001*, *sens_abstol=0.01*)
Checks for almost-equality between two solutions

static decompress_file (*file*)
Load a gzip-compressed pickle file

diff (*other*, *showvars=None*, *, *constraintsdiff=True*, *senssdiff=False*, *sensstol=0.1*, *absdiff=False*, *abstol=0*, *reldiff=True*, *reltol=1.0*, ***tableargs*)
Outputs differences between this solution and another

other [solution or string] strings will be treated as paths to pickled solutions

senssdiff [boolean] if True, show sensitivity differences

sensstol [float] the smallest sensitivity difference worth showing

abssdiff [boolean] if True, show absolute differences

absstol [float] the smallest absolute difference worth showing

reldiff [boolean] if True, show relative differences

reltol [float] the smallest relative difference worth showing

str

modelstr = ''

name_collision_varkeys ()
Returns the set of contained varkeys whose names are not unique

plot (*posys=None*, *axes=None*)
Plots a sweep for each posy

save (*filename='solution.pkl'*, ***pickleargs*)
Pickles the solution and saves it to a file.

Solution can then be loaded with e.g.: >>> import pickle >>> pickle.load(open("solution.pkl"))

save_compressed (*filename='solution.pgz'*, ***cpickleargs*)
Pickle a file and then compress it into a file with extension.

savecsv (*showvars=None*, *filename='solution.csv'*, *valcols=5*)
Saves primal solution as a CSV sorted by modelname, like the tables.

savemat (*filename='solution.mat'*, *showvars=None*, *excluded=('unnecessary lineage', 'vec')*)
Saves primal solution as matlab file

savetxt (*filename='solution.txt'*, *printmodel=True*, ***kwargs*)
Saves solution table as a text file

subinto (*posy*)
Returns NomialArray of each solution substituted into posy.

summary (*showvars=()*, *ntopsenss=5*, ***kwargs*)
Print summary table, showing top sensitivities and no constants

table (*showvars=(), tables=('cost', 'warnings', 'sweepvariables', 'freevariables', 'constants', 'sensitivities', 'tightest constraints'), **kwargs*)
A table representation of this SolutionArray

tables: Iterable

Which to print of ("cost", "sweepvariables", "freevariables", "constants", "sensitivities")

fixedcols: If true, print vectors in fixed-width format latex: int

If > 0, return latex format (options 1-3); otherwise plain text

included_models: Iterable of strings If specified, the models (by name) to include

excluded_models: Iterable of strings If specified, model names to exclude

str

table_titles = {'constants': 'Fixed Variables', 'freevariables': 'Free Variables

todataframe (*showvars=None, excluded=('unnecessary lineage', 'vec')*)
Returns primal solution as pandas dataframe

varnames (*showvars, exclude*)
Returns list of variables, optionally with minimal unique names

gpkit.solution_array.**cast** (*function, val1, val2*)
Relative difference between val1 and val2 (positive if val2 is larger)

gpkit.solution_array.**constraint_table** (*data, title, sortbymodel=True, showmodels=True, **_*)
Creates lines for tables where the right side is a constraint.

gpkit.solution_array.**insenss_table** (*data, _, maxval=0.1, **kwargs*)
Returns insensitivity table lines

gpkit.solution_array.**loose_table** (*self, _, min_senss=1e-05, **kwargs*)
Return constraint tightness lines

gpkit.solution_array.**senss_table** (*data, showvars=(), title='Variable Sensitivities', **kwargs*)
Returns sensitivity table lines

gpkit.solution_array.**tight_table** (*self, _, ntightconstrs=5, tight_senss=0.01, **kwargs*)
Return constraint tightness lines

gpkit.solution_array.**topsenss_filter** (*data, showvars, nvars=5*)
Filters sensitivities down to top N vars

gpkit.solution_array.**topsenss_table** (*data, showvars, nvars=5, **kwargs*)
Returns top sensitivity table lines

gpkit.solution_array.**unrolled_absmax** (*values*)
From an iterable of numbers and arrays, returns the largest magnitude

gpkit.solution_array.**var_table** (*data, title, *, printunits=True, latex=False, rawlines=False, varfmt='%s : ', valfmt='%-.4g ', vecfmt='%-8.3g', minval=0, sortbyvals=False, hidebelowminval=False, included_models=None, excluded_models=None, sortbymodel=True, maxcolumns=5, skipifempty=True, **_*) 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 printunits : bool latex : int

If > 0, return latex format (options 1-3); otherwise plain text

varfmt [string] format for variable names

valfmt [string] format for scalar values

vecfmt [string] format for vector values

minval [float] skip values with all(abs(value)) < minval

sortbyvals [boolean] If true, rows are sorted by their average value instead of by name.

included_models [Iterable of strings] If specified, the models (by name) to include

excluded_models [Iterable of strings] If specified, model names to exclude

`gpkit.solution_array.`**`warnings_table`**(*self*, *_*, ***kwargs*)
Makes a table for all warnings in the solution.

10.1.11 gpkit.units module

wraps pint in gpkit monomials

class `gpkit.units.`**`GPkitUnits`**
Bases: `object`

Return Monomials instead of Quantitites

`division_cache = {}`

`monomial_cache = {}`

`multiplication_cache = {}`

`of_division`(*numerator*, *denominator*)
Cached unit division. Requires Quantity inputs.

`of_product`(*thing1*, *thing2*)
Cached unit division. Requires united inputs.

`gpkit.units.`**`qty`**(*unit*)
Returns a Quantity, caching the result for future retrievals

10.1.12 gpkit.varkey module

Defines the VarKey class

class `gpkit.varkey.`**`VarKey`**(*name=None*, ***descr*)
Bases: *`gpkit.repr_conventions.ReprMixin`*

An object to correspond to each 'variable name'.

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

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

VarKey with the given name and descr.

latex(*excluded=()*)
Returns latex representation.

models
Returns a tuple of just the names of models in self.lineage

str_without(*excluded=()*)
Returns string without certain fields (such as 'lineage').

subscripts = ('lineage', 'idx')

classmethod unique_id()
Increment self.count and return it

vars_of_a_name = {}

10.1.13 Module contents

GP and SP modeling package

CHAPTER 11

Citing GPkit

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

```
@inproceedings{burnell2020gpkit,
    author={Burnell, Edward and Damen, Nicole B and Hoburg, Warren},
    title={\hbox{GPkit}: A Human-Centered Approach to Convex Optimization in Engineering Design},
    booktitle={Proceedings of the 2020 {CHI} Conference on Human Factors in Computing Systems},
    year={2020},
    doi={10.1145/3313831.3376412}
}
```

(and you can read that paper, which describes some of GPkit's design philosophy, here.)

CHAPTER 12

Acknowledgements

We thank the following contributors for helping to improve GPkit:

- Marshall Galbraith for setting up continuous integration.
- Stephen Boyd for inspiration and suggestions.
- Kirsten Bray for designing the GPkit logo.

CHAPTER 13

Release Notes

Release notes are available on Github

Python Module Index

g

Index

A

B

C

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W