gpkit Documentation

Release 1.1

MIT Department of Aeronautics and Astronautics

Jan 09, 2022

Contents

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

Our hopes are to bring geometric programming into engineering design processes in a disciplined and collaborative way, and to encourage research with GPs by providing an extensible object-oriented framework.

GPkit abstracts away solvers so 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:

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

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

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

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

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

For example, the expressions

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

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

written as:

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

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

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

1.2 Why are GPs special?

Geometric programs have several powerful properties:

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

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

1.3 What are Signomials / Signomial Programs?

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

A Signomial Program has signomial constraints. 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 9 solver with `pip install Mosek`, then a license as described below
3. (optional) Install the MOSEK 8 solver as described below
4. Run `pip install gpkit` in the appropriate terminal or command prompt.
5. 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 8

GPkit interfaces with two off the shelf solvers: cvxopt, and MOSEK (versions 8 and 9). Cvxopt is open source and installed by default; MOSEK requires a commercial licence or (free) academic license. In MOSEK version 8 GPkit uses the command-line interface `mskexpopt` solver, while in MOSEK 9 it uses the more active exponential-cone interface (and hence supports *Choice Variables*).

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

- **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.
- **Download MOSEK 8, then:**
  - Follow these steps for Windows.
- Request an academic license file and put it in `C:\Users\(your_username)\mosek\`

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 %.4g." % 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()

x_min = Variable("x_min", 2, "ft")
```

(continues on next page)

(continued from previous page)

```
m4 = Model(x, [x >= x_min])
m4.debug()
```

```
<DEBUG> Model is feasible with these modifications:

Arbitrarily Bounded Variables
-----------------------------
   value near upper bound of 1e+30: y
 sensitive to upper bound of 1e+30: y

Relaxed Constants
-----------------
  x_min [ft]: relaxed from 2 to 1

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

<DEBUG> Model is not feasible with relaxed constants and bounded variables.
<DEBUG> Model is feasible with these modifications:

Relaxed Constraints
-------------------
   1: 3500% relaxed, from    x [ft] >= 1 [yd]
                       to 36·x [ft] >= 1 [yd]

# And one that's only unbounded

<DEBUG> Model is feasible with these modifications:

Arbitrarily Bounded Variables
-----------------------------
   value near upper bound of 1e+30: y
 sensitive to upper bound of 1e+30: y

<DEBUG> Model seems feasible without modification, or only needs relaxations␣
↪of less than 1%. Check the returned solution for details.
```

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 'optimal**

  – 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 'optimal**

  – 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 Run**

  - 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 thar`**

  - 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:

```
"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 of 1e+30"]
        == sol["boundedness"]["sensitive to upper bound of 1e+30"])
```

Upon viewing the printed output,

```

```

(continues on next page)

(continued from previous page)

```
    Cost
 (1e-30) 1/x
          (1e-30)



 Model
       x  1e+30



~~~~~~~~
WARNINGS
~~~~~~~~
Arbitrarily Bounded Variables
-----------------------------
   value near upper bound of 1e+30: x
 sensitive to upper bound of 1e+30: x
~~~~~~~~

Free Variables
--------------
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
    ]

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
from gpkit.breakdowns import Breakdowns
Breakdowns(mr1.solution).trace("cost")
```

(continues on next page)

(continued from previous page)

```
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)
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 Function
-------------
 x

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

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

Cost Function
-------------
 C

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

(continues on next page)

(continued from previous page)

```
   Cost
 (1.41) x_min
         (2, fixed)




       x  C·x_max


       x_max = 1
 Model

       x_min = 2


       x_min  C·x


~~~~~~~~
WARNINGS
~~~~~~~~
Relaxed Constraints
-------------------
All constraints relaxed by 42%
~~~~~~~~

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

  | Relax
C : 1.414

Fixed Variables
---------------
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


C (1.41)
  breaks down into:
```

(continues on next page)

(continued from previous page)

```
    C (1.41)
      which in: x  C·x_max (sensitivity +0.5)
      { through a factor of 1/x_max (1, fixed) }
      breaks down into:
        x (1.41)
          which in: x_min  C·x (sensitivity +0.5)
          breaks down into:
            { through a factor of 1/C (0.707) }
            x_min (2, fixed)

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

Cost Function
-------------
 C[:].prod()·x^0.01

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


 Cost
  (2) 1/x
       (1)




       x_min = 2



       x_min  C[1]·x
 Model


       x  C[0]·x_max



       x_max = 1



~~~~~~~~
WARNINGS
~~~~~~~~
Relaxed Constraints
-------------------
   1:  100% relaxed, from     x >= x_min
                        to x_min <= 2·x
```

(continues on next page)

(continued from previous page)

```
~~~~~~~~

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


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

Cost Function
-------------
 [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  OriginalValues.x_max/Relax2.x_max
      x_max  OriginalValues.x_max·Relax2.x_max
    "x_min":
      Relax2.x_min  1
      x_min  OriginalValues.x_min/Relax2.x_min
      x_min  OriginalValues.x_min·Relax2.x_min


 Cost
  (2) 1/Relax2.x_min
       (0.5)
```

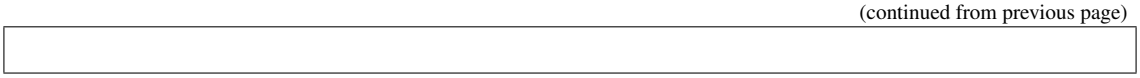
(continues on next page)

(continued from previous page)

```
       x  x_min


       x_min = 1


 Modelx_min  OriginalValues.x_min/Relax2.x_min


       x  x_max


       x_max = 1


       x_max  OriginalValues.x_max·Relax2.x_max



~~~~~~~~
WARNINGS
~~~~~~~~
Relaxed Constants
-----------------
  x_min: relaxed from 2 to 1
~~~~~~~~

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  OriginalValues.x_min/Relax2.x_min
 +0.99 : x  x_max
 +0.99 : x_max  OriginalValues.x_max·Relax2.x_max
```

(continues on next page)

(continued from previous page)



CHAPTER 5

Visualization and Interaction

Code in this section uses the CE solar model except where noted otherwise.

5.1 Model and Variable Breakdowns

Model breakdowns (similar to the sankey diagrams below) show the hierarchy of a model scaled by the sensitivity of its constraints and fixed variables.

Variable breakdowns show how a variable "breaks down" into smaller expressions. For example if the constraint `x_total >= x1 + x2` is tight (that is, has a sensitivity greater than zero, indicating that the right hand side is "pushing" against the left), then `x_total` can be said to "break down" into `x1` and `x2`, each of which may have their own breakdowns. If multiple constraints break down a variable, the most sensitive one is chosen; if none do, than constraints such as `1 >= x1/x_total + x2/x_total` will be rearranged in an attempt to create a valid breakdown constraint like that above.

```
"An example to show off Breakdowns"
import os
import pickle
import pint
from packaging import version
from gpkit.breakdowns import Breakdowns

if version.parse(pint.__version__) >= version.parse("0.9"):
    # the code to create solar.p is in ./breakdowns/solartest.py
    filepath = os.path.dirname(os.path.realpath(__file__)) + os.sep + "solar.
↪p"
    sol = pickle.load(open(filepath, "rb"))
    bds = Breakdowns(sol)

    print("Cost breakdown (as seen in solution tables)")
    print("==============")
    bds.plot("cost")
```

(continues on next page)

(continued from previous page)

```
    print("Variable breakdowns (note the two methods of access)")
    print("==================")
    varkey, = sol["variables"].keymap[("Mission.FlightSegment.AircraftPerf"
                                       ".AircraftDrag.Poper")]
    bds.plot(varkey)
    bds.plot("AircraftPerf.AircraftDrag.MotorPerf.Q")

    print("Combining the two above by increasing maxwidth")
    print("----------------------------------------------")
    bds.plot("AircraftPerf.AircraftDrag.Poper", maxwidth=105)

    print("Model sensitivity breakdowns (note the two methods of access)")
    print("============================")
    bds.plot("model sensitivities")
    bds.plot("Aircraft")

    print("Exhaustive variable breakdown traces (and configuration arguments)
↪")
    print("====================================")
    # often useful as a reference point when reading traces
    bds.plot("AircraftPerf.AircraftDrag.Poper", height=12)
    # includes factors, can be useful for reading traces as well
    bds.plot("AircraftPerf.AircraftDrag.Poper", showlegend=True)
    print("\nPermissivity = 2 (the default)")
    print("----------------")
    bds.trace("AircraftPerf.AircraftDrag.Poper")
    print("\nPermissivity = 1 (stops at Pelec = v·i)")
    print("----------------")
    bds.trace("AircraftPerf.AircraftDrag.Poper", permissivity=1)

    # you can also produce Plotly treemaps/icicle plots of your breakdowns
    fig = bds.treemap("model sensitivities", returnfig=True)
    fig = bds.icicle("cost", returnfig=True)
    # uncommenting any of the below makes and shows the plot directly
    # bds.icicle("model sensitivities")
    # bds.treemap("cost")
```

```
Cost breakdown (as seen in solution tables)
==============









                    Battery.W Battery.E
                     (370lbf)   (165,913kJ)


     Cost
 (699lbf) Wtotal
           (699lbf)
                              Wing.BoxSpar.W
                    Wing.W     (96.1lbf)
                     (139lbf) Wing.WingSecondStruct.W
                    Motor.W
```

(continues on next page)

(continued from previous page)

```
                    SolarCells.W
                    Empennage.W
                    Wavn
                    [6 terms]

Variable breakdowns (note the two methods of access)
====================









                                                                  MotorPerf.Q
 AircraftPerf.AircraftDrag.PoperMotorPerf.Pelec MotorPerf.i   (4.8N·m)
                         (3,194W)   (0.685kW)        (36.8A)



                                                                 i0
                                                                  (4.5A, fixed)
                                  Pavn
                                  Ppay



 AircraftPerf.AircraftDrag.MotorPerf.Q
                               (4.8N·m) ..ActuatorProp.CP
                                         (0.00291)

Combining the two above by increasing maxwidth
----------------------------------------------









                                                                  MotorPerf.Q␣
↪ActuatorProp.CP
 AircraftPerf.AircraftDrag.PoperMotorPerf.Pelec MotorPerf.i   (4.8N·m)     (0.
↪00291)
                         (3,194W)   (0.685kW)        (36.8A)



                                                                 i0
                                                                  (4.5A, fixed)
                                  Pavn
                                  Ppay

Model sensitivity breakdowns (note the two methods of access)
=============================
```

(continues on next page)

(continued from previous page)

```
                                                                ActuatorProp

                              AircraftPerf AircraftDrag MotorPerf

      Mission FlightSegment                                [17 terms]

                              FlightState
 Model                       GustL
                              SteadyLevelFlight
                              [49 terms]
               Climb



      Aircraft


      g = 9.81m/s²


                  etadischarge = 0.98

                  W  E·minSOC/hbatt/etaRTE/etapack·g
                  etaRTE = 0.95
         Battery etapack = 0.85
                  hbatt = 350W·hr/kg
                  minSOC = 1.03


 Aircraft
         Wing


         Wtotal/mfac  Fuselage.W[0,0] + Fuselage.W[1,0] + Fuselage.W[2,0]␣
↪...

         mfac = 1.05

         Empennage
         [23 terms]


Exhaustive variable breakdown traces (and configuration arguments)
==================================




 AircraftPerf.AircraftDrag.PoperMotorPerf.Pelec MotorPerf.i MotorPerf.Q
                       (3,194W)   (0.685kW)        (36.8A)      (4.8N·m)



                                                                i0
```

(continues on next page)

(continued from previous page)

```
                                  Pavn
                                   ||||
                                   ||||
                                   ||||
                                   ||||
                                   ||||
                                   ||DCBAActuatorProp.CP
 AircraftPerf.AircraftDrag.PoperHGFE|| (0.00291)
                          (3,194W) J|||
                                   ||||
                                   ||||
                                   ||||
                                   ||i0 (4.5A, fixed)

                                  Pavn (200W, fixed)
                                  Ppay (100W, fixed)

 A   4.53e-05·FlightState.rho·ActuatorProp.omega²·Propeller.R⁵   ×1,653N·m
↪ [free factor]
 B   ActuatorProp.Q                                              = 4.8N·m
 C   MotorPerf.Q                                                 = 4.8N·m
 D   Kv                                                           ×64.2rpm/V
↪  [free factor]
 E   MotorPerf.i                                                 = 36.8A
 F   MotorPerf.v                                                  ×18.6V
↪  [free factor]
 G   MotorPerf.Pelec                                             = 0.685kW
 H   Nprop                                                        ×4, fixed
 J   mpower                                                       ×1.05, fixed

Permissivity = 2 (the default)
----------------

AircraftPerf.AircraftDrag.Poper (3,194W)
  which in: Poper/mpower  Pavn + Ppay + Pelec·Nprop (sensitivity +5.6)
  { through a factor of AircraftPerf.AircraftDrag.mpower (1.05, fixed) }
  breaks down into 3 monomials:
    1) forming 90% of the RHS and 90% of the total:
      { through a factor of Nprop (4, fixed) }
      AircraftPerf.AircraftDrag.MotorPerf.Pelec (0.685kW)
        which in: Pelec = v·i (sensitivity -5.1)
        breaks down into:
          { through a factor of AircraftPerf.AircraftDrag.MotorPerf.v (18.
↪6V) }
          AircraftPerf.AircraftDrag.MotorPerf.i (36.8A)
            which in: i  Q·Kv + i0 (sensitivity +5.4)
            breaks down into 2 monomials:
              1) forming 87% of the RHS and 79% of the total:
                { through a factor of Kv (64.2rpm/V) }
                AircraftPerf.AircraftDrag.MotorPerf.Q (4.8N·m)
                  which in: Q = Q (sensitivity -4.7)
                  breaks down into:
                    AircraftPerf.AircraftDrag.ActuatorProp.Q (4.8N·m)
                      which in: CP  Q·omega/(0.5·rho·(omega·R)³·π·R²)
↪(sensitivity +4.7)
```

(continues on next page)

(continued from previous page)

```
                    { through a factor of 4.53e-05·FlightState.
↪rho·AircraftPerf.AircraftDrag.ActuatorProp.omega²·Propeller.R⁵ (1,653N·m) }
                    breaks down into:
                      AircraftPerf.AircraftDrag.ActuatorProp.CP (0.00291)
              2) forming 12% of the RHS and 11% of the total:
                i0 (4.5A, fixed)
    2) forming 6% of the RHS and 6% of the total:
      AircraftPerf.AircraftDrag.Pavn (200W, fixed)
    3) forming 3% of the RHS and 3% of the total:
      AircraftPerf.AircraftDrag.Ppay (100W, fixed)

Permissivity = 1 (stops at Pelec = v·i)
----------------

AircraftPerf.AircraftDrag.Poper (3,194W)
  which in: Poper/mpower  Pavn + Ppay + Pelec·Nprop (sensitivity +5.6)
  { through a factor of AircraftPerf.AircraftDrag.mpower (1.05, fixed) }
  breaks down into 3 monomials:
    1) forming 90% of the RHS and 90% of the total:
      { through a factor of Nprop (4, fixed) }
      AircraftPerf.AircraftDrag.MotorPerf.Pelec (0.685kW)
        which in: Pelec = v·i (sensitivity -5.1)
        breaks down into:
          AircraftPerf.AircraftDrag.MotorPerf.i·AircraftPerf.AircraftDrag.
↪MotorPerf.v (685A·V)
    2) forming 6% of the RHS and 6% of the total:
      AircraftPerf.AircraftDrag.Pavn (200W, fixed)
    3) forming 3% of the RHS and 3% of the total:
      AircraftPerf.AircraftDrag.Ppay (100W, fixed)
```

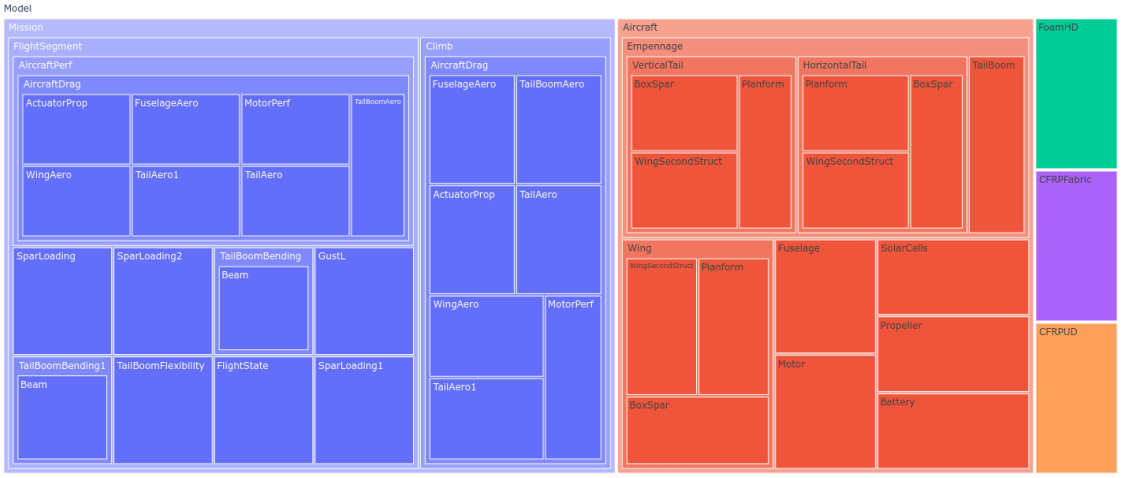
If permissivity is greater than 1, the breakdown will always proceed if a breakdown variable is available in the monomial, and will choose the most sensitive one if multiple are available. If permissivity is 1, breakdowns will stop when there are multiple breakdown variables multiplying each other. If permissivity is 0, breakdowns will stop when any free variables multiply each other. If permissivity is between 0 and 1, it will follow the behavior for 1 if the monomial represents a fraction of the total greater than `1 - permissivity`, and the behavior for 0 otherwise.

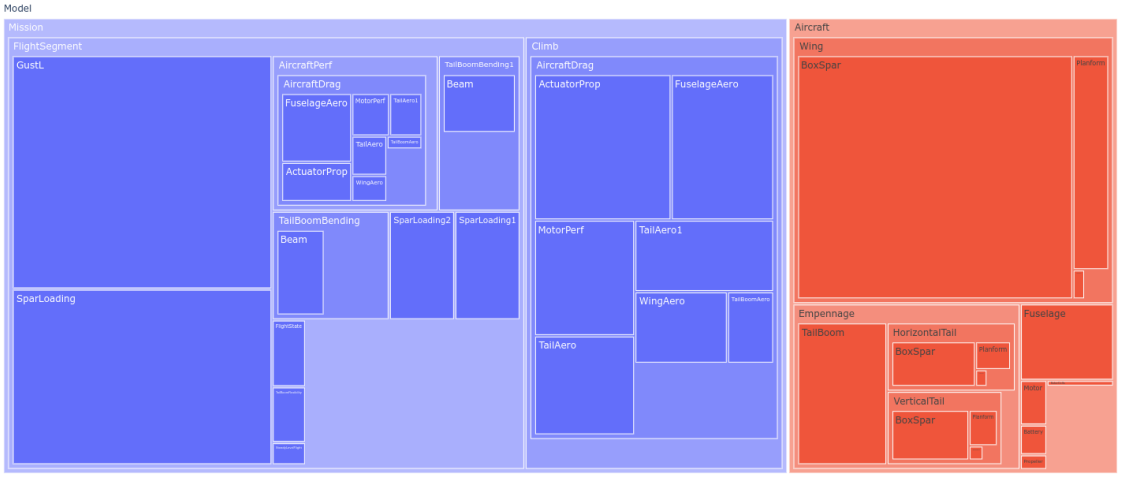
5.2 Model Hierarchy Treemaps

```
import plotly
from gpkit.interactive.plotting import treemap
from solar.solar import *
Vehicle = Aircraft(Npod=3, sp=True)
M = Mission(Vehicle, latitude=[20])
fig = treemap(M)
plotly.offline.plot(fig, filename="treemap.html")
```

and, using sizing and counting by constraints instead of variables (the default):

```
fig = treemap(M, itemize="constraints", sizebycount=True)
plotly.offline.plot(fig, filename="sizedtreemap.html")
```


Model
Mission
FlightSegment
AircraftPerf
AircraftDrag
ActuatorProp
FuselageAero
MotorPerf
TailBoomAero
WingAero
TailAero1
TailAero
SparLoading
SparLoading2
TailBoomBending
Beam
GustL
TailBoomBending1
Beam
TailBoomFlexibility
FlightState
SparLoading1
Climb
AircraftDrag
FuselageAero
TailBoomAero
ActuatorProp
TailAero
WingAero
MotorPerf
TailAero1
Aircraft
Empennage
VerticalTail
BoxSpar
Planform
WingSecondStruct
HorizontalTail
Planform
BoxSpar
WingSecondStruct
TailBoom
Wing
WingSecondStruct
Planform
BoxSpar
Fuselage
Motor
SolarCells
Propeller
Battery
FoamHD
CFRPFabric
CFRPUD


Model
Mission
FlightSegment
GustL
SparLoading
AircraftPerf
AircraftDrag
FuselageAero
MotorPerf
TailAero1
TailAero
ActuatorProp
WingAero
TailBoomBending1
Beam
TailBoomBending
Beam
SparLoading2
SparLoading1
Climb
AircraftDrag
ActuatorProp
FuselageAero
MotorPerf
TailAero1
WingAero
TailAero
Aircraft
Wing
BoxSpar
Planform
Empennage
TailBoom
HorizontalTail
BoxSpar
Planform
VerticalTail
BoxSpar
Fuselage

5.3 Variable Reference Plots

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

from gpkit.interactive.references import referencesplot
referencesplot(M, openimmediately=True)
```

Running the code above will produce two files in your working directory: `referencesplot.html` and `referencesplot.json`, and (unless you specify `openimmediately=False`) open the former in your web browser, showing you something like this:

Click to see the interactive version of this plot.

When a model's name is hovered over its connections are highlighted, showing in red the other models it imports variables from to use in its constraints and in blue the models that import variables from it.

By default connections are shown with equal width ("Unweighted"). When "Global Sensitivities" is selected, connection width is proportional to the sensitivity of all variables in that connection to the importing model, corresponding exactly to how much the model's cost would decrease if those variables were relaxed in only that importing model. This can give a sense of which connections are vital to the overall model. When "Normalized Sensitivities" is selected, that global weight is divided by the weight of all variables in the importing model, giving a sense of which connections are vital to each subsystem.

5.4 Sensitivity Diagrams

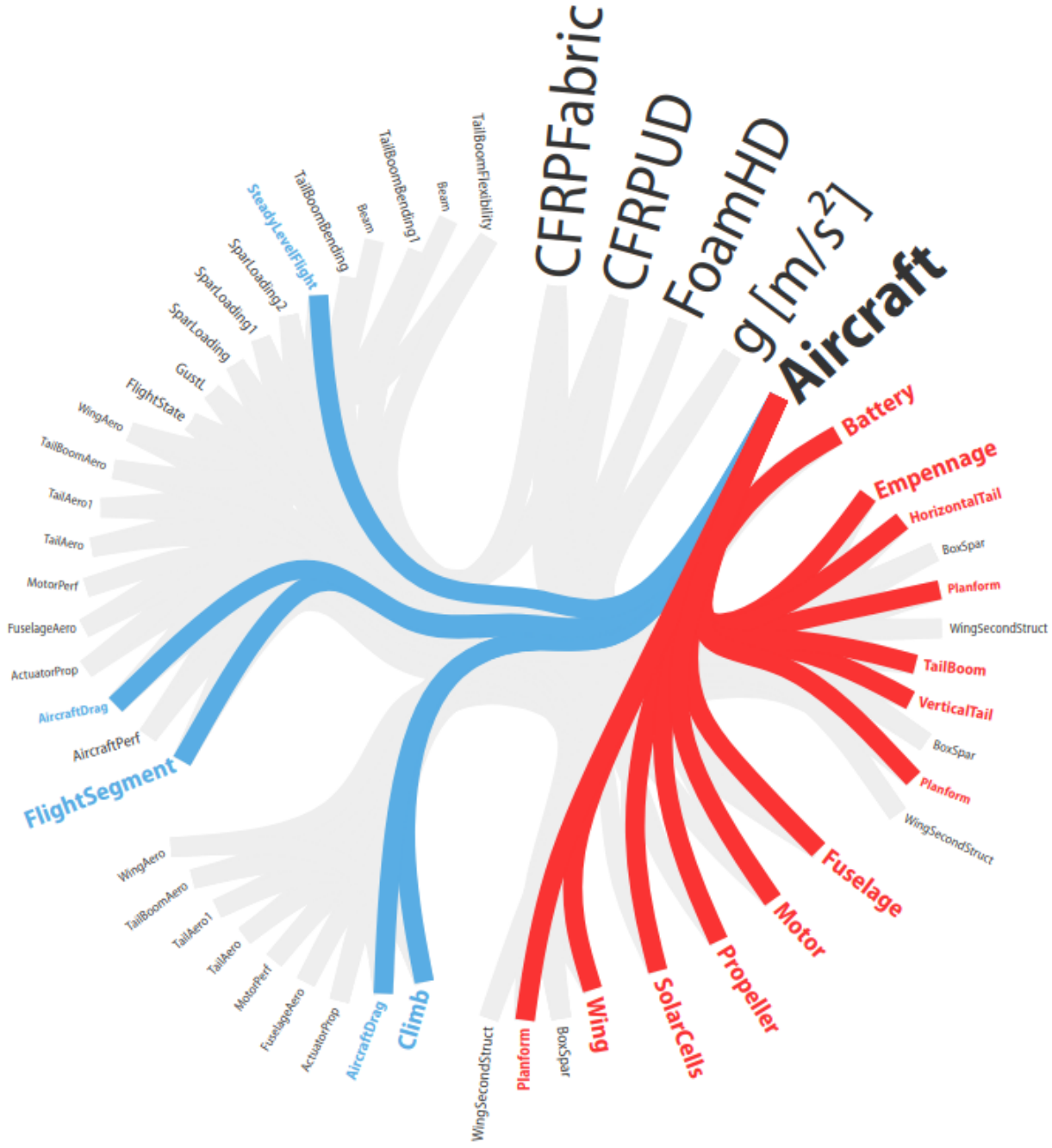
5.4.1 Requirements

- 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`

5.4.2 Example

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

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:


CFRPFabric
CFRPUD
FoamHD
g [m/s²]
Aircraft
Battery
Empennage
HorizontalTail
BoxSpar
Planform
WingSecondStruct
TailBoom
VerticalTail
BoxSpar
Planform
WingSecondStruct
Fuselage
Motor
Propeller
SolarCells
Wing
Planform
BoxSpar
WingSecondStruct
TailBoomFlexibility
TailBoomBending
Beam
TailBoomBending1
Beam
SteadyLevelFlight
SparLoading2
SparLoading1
SparLoading
GustL
FlightState
WingAero
TailBoomAero
TailAero1
TailAero
MotorPerf
FuselageAero
ActuatorProp
AircraftDrag
AircraftPerf
FlightSegment
WingAero
TailBoomAero
TailAero1
TailAero
MotorPerf
FuselageAero
ActuatorProp
AircraftDrag
Climb

5.4.3 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.

5.4.4 Usage

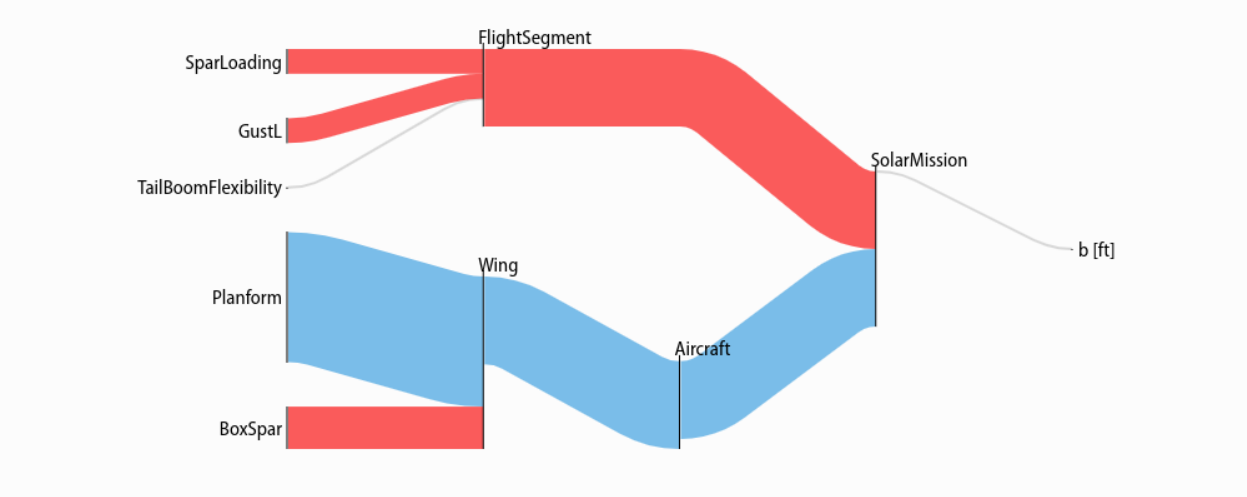
Variables

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):

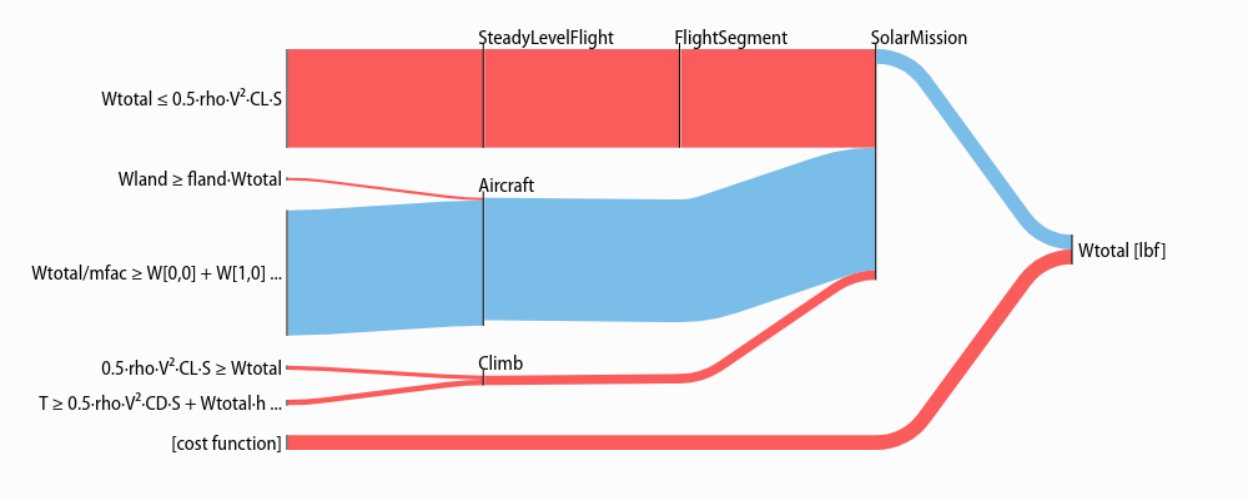
```
Sankey(sol, M, "SolarMission").diagram(M.aircraft.wing.planform.b,␣
↪showconstraints=False)
```



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:

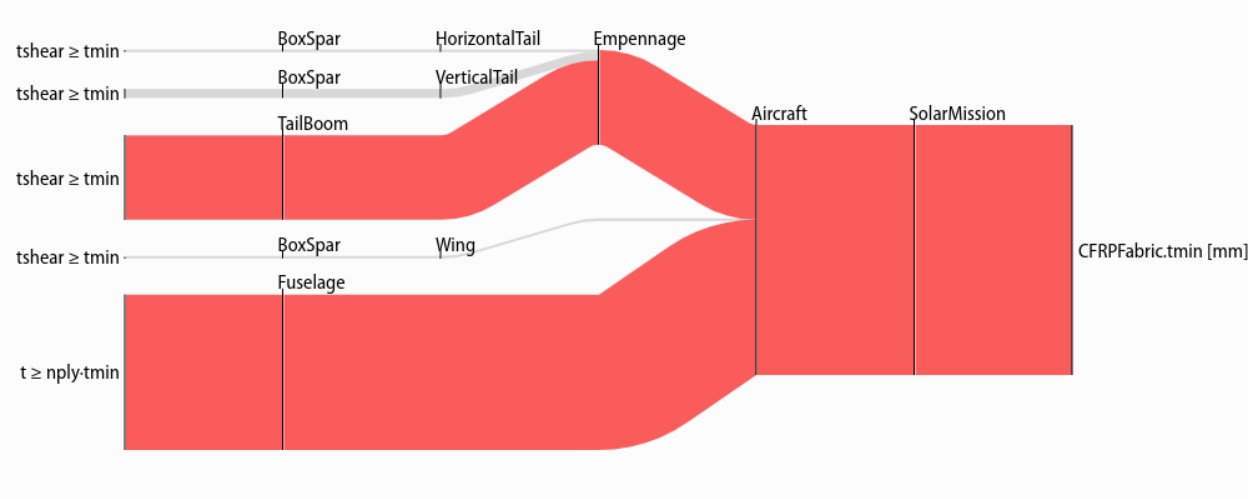
```
Sankey(sol, M, "SolarMission").diagram(M.aircraft.Wtotal)
```



Fixed

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

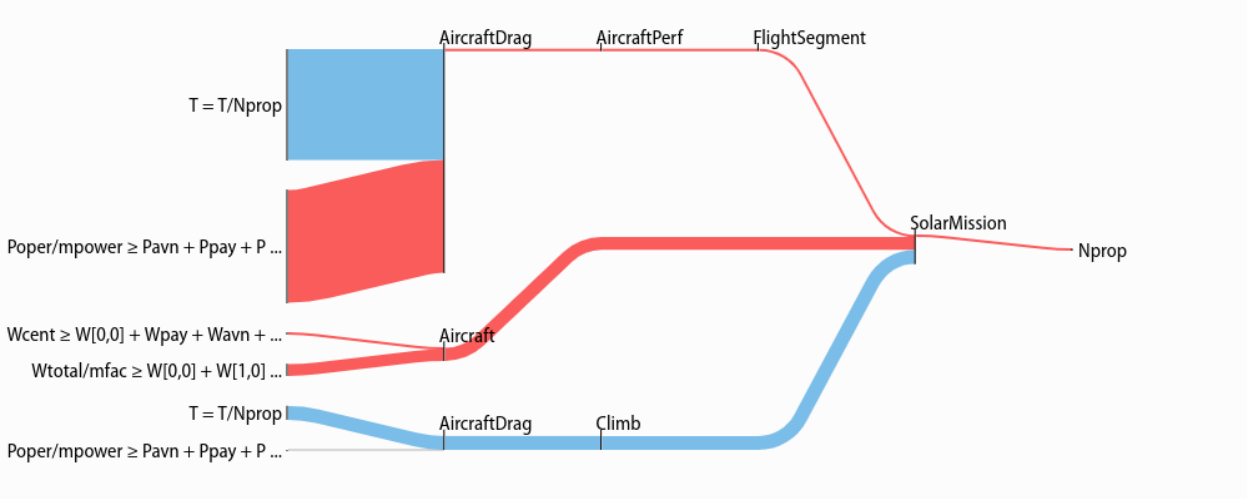
```
Sankey(sol, M, "SolarMission").diagram(M.variables_byname("tmin")[0],␣
↪left=100)
```



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)
```

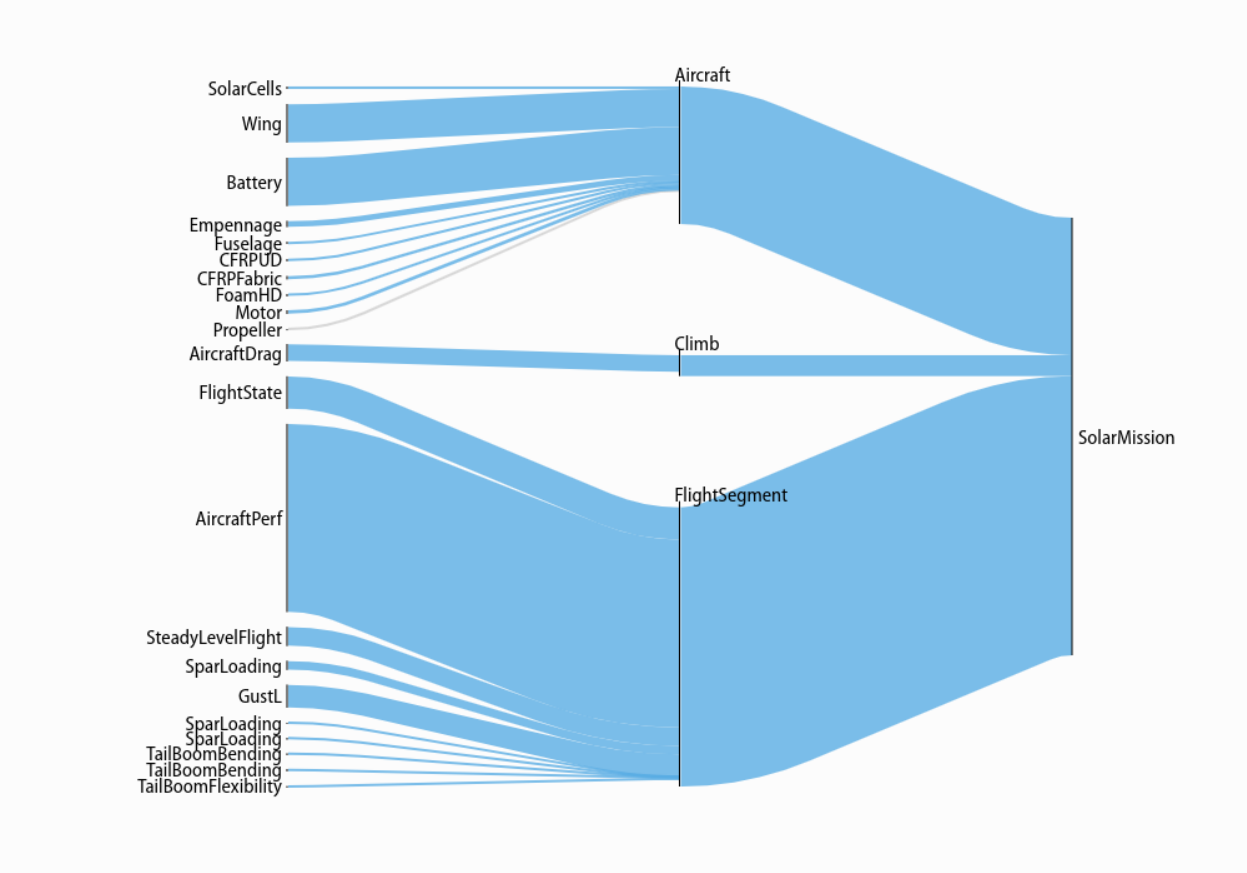
The only piece of unexplained syntax in this is `minsenss`. Perhaps unsurprisingly, this just limits the links shown to only those whose sensitivity exceeds that minimum; it's quite useful for exploring a large model.

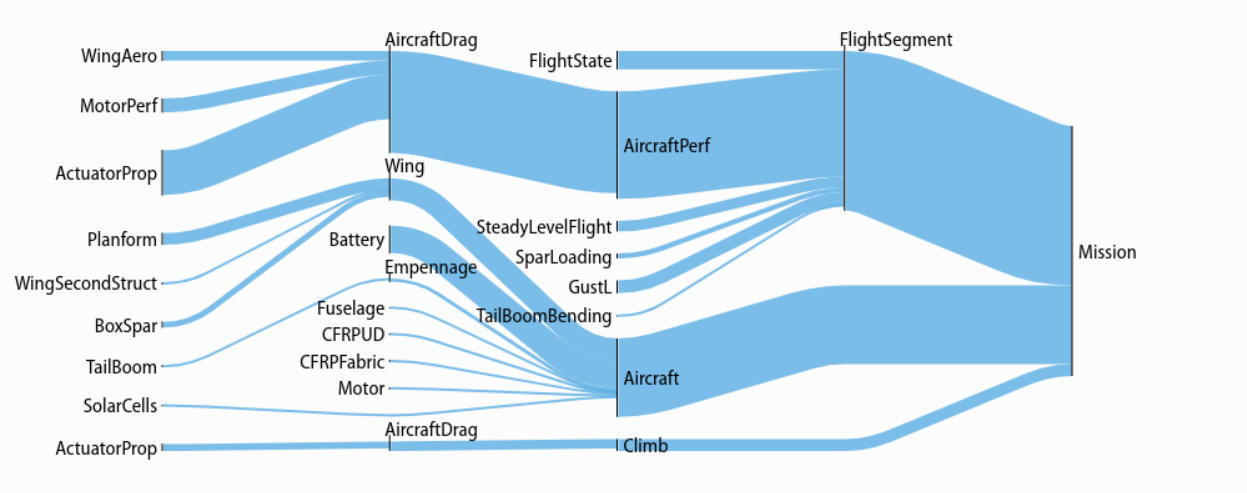
5.5 Plotting a 1D Sweep

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)

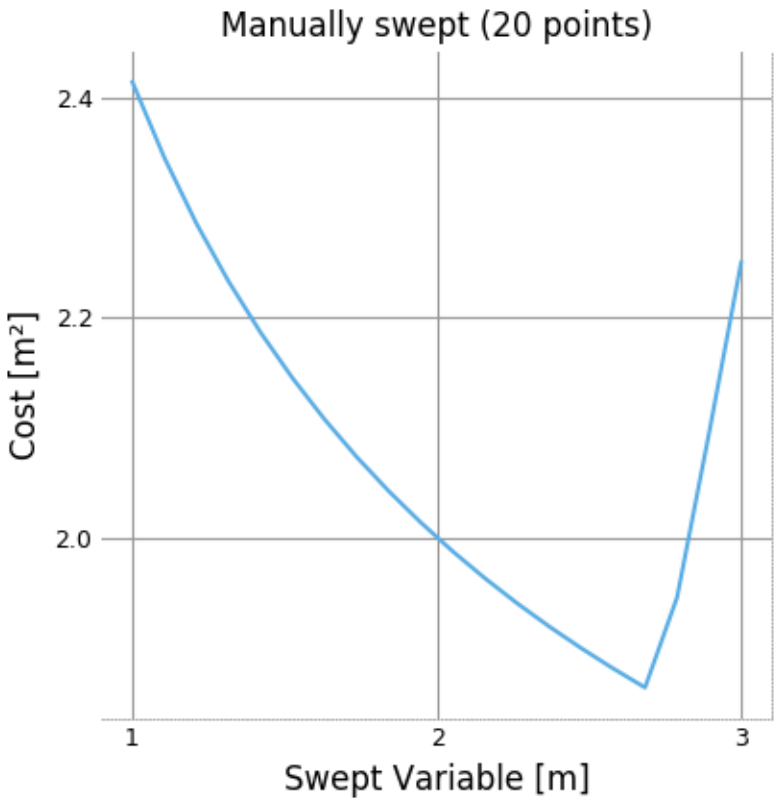

SolarCells
Wing
Battery
Empennage
Fuselage
CFRPUD
CFRPFabric
FoamHD
Motor
Propeller
AircraftDrag
FlightState
AircraftPerf
SteadyLevelFlight
SparLoading
GustL
SparLoading
SparLoading
TailBoomBending
TailBoomBending
TailBoomFlexibility
Aircraft
Climb
FlightSegment
SolarMission

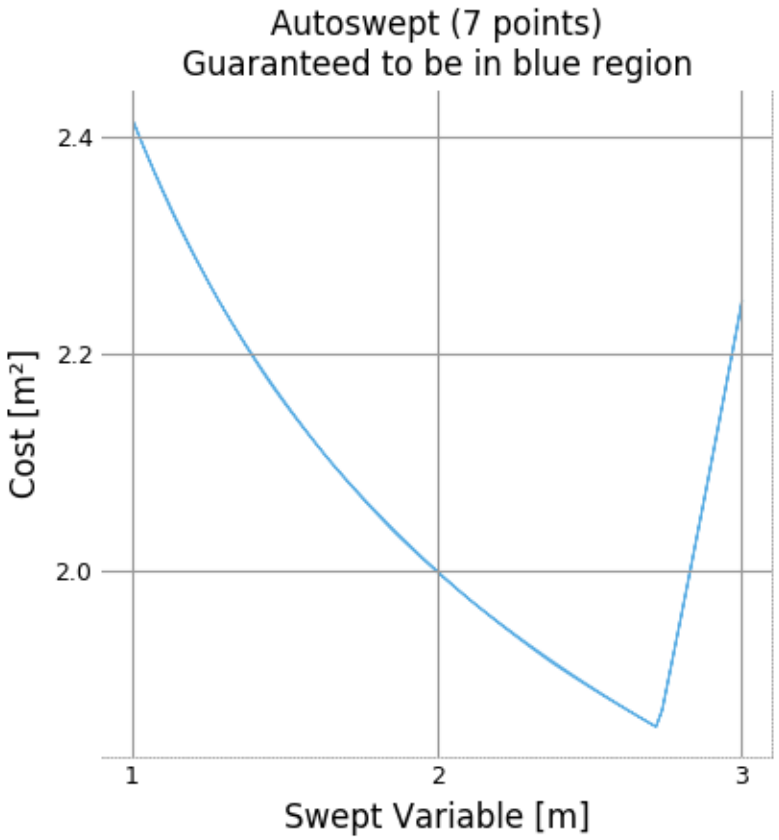

WingAero
MotorPerf
ActuatorProp
Planform
WingSecondStruct
BoxSpar
TailBoom
SolarCells
ActuatorProp
AircraftDrag
Wing
Battery
Empennage
Fuselage
CFRPUD
CFRPFabric
Motor
AircraftDrag
FlightState
AircraftPerf
SteadyLevelFlight
SparLoading
GustL
TailBoomBending
Aircraft
Climb
FlightSegment
Mission

(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
2.4
2.2
2.0
Cost [m²]
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


 Cost
```

(continues on next page)

(continued from previous page)

```
  (1) 1



 Model
       x  1


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

__________

VECTORIZED


 Cost
  (2) 2




              x[0]  1

       Test1
 Model
              x[2]  1


       x[1]  2


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

6.5 Multipoint analysis modeling

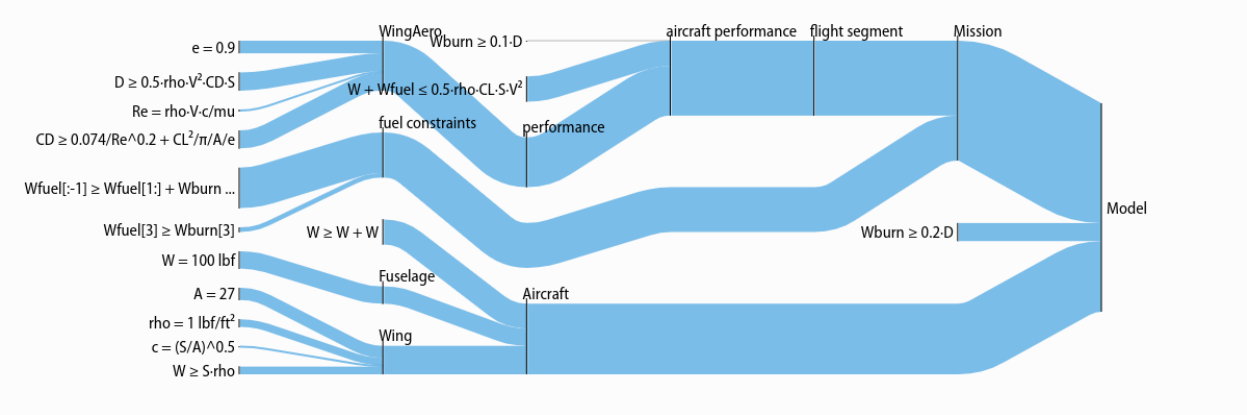
In many engineering models, there is a physical object that is operated in multiple conditions. Some variables correspond to the design of the object (size, weight, construction) while others are vectorized over the different conditions (speed, temperature, altitude). By combining named models and vectorization 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):




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


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
```

(continues on next page)

(continued from previous page)

```
        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

    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
```

(continues on next page)

(continued from previous page)

```
    ---------------
    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)

        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
```

(continues on next page)

(continued from previous page)

```
    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())
    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):
```

(continues on next page)

(continued from previous page)

```
    """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

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))

try:
    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)
    # the line below shows an interactive graph if run in jupyter notebook
    sankey  # pylint: disable=pointless-statement
except (ImportError, ModuleNotFoundError):
    print("Making Sankey diagrams requires the ipysankeywidget package")

from gpkit.interactive.references import referencesplot
referencesplot(M, openimmediately=False)
```

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

```

Cost Function
-------------
 Wfuel[0]
```

(continues on next page)

(continued from previous page)

```
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·Mission.FlightSegment.FlightState.
↪rho[:]·CL[:]·S·V[:]^2
    "performance":
      WingAero
       D[:]  0.5·Mission.FlightSegment.FlightState.rho[:]·V[:]^2·CD[:]·S
       Re[:] = Mission.FlightSegment.FlightState.rho[:]·V[:]·c/mu[:]
       CD[:]  0.074/Re[:]^0.2 + CL[:]^2/π/A/e[:]

   FlightState
    (no constraints)

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

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



                                                Wburn[2]   CD[2]
                                                 (0.272lbf)  (0.0189)

                                     Wfuel[2]
                                      (0.544lbf)
                         Wfuel[1]                Wfuel[3]   CD[3]
                          (0.817lbf)               (0.272lbf)  (0.0188)
      Cost
 (1.09lbf) Wfuel[0]
             (1.09lbf)                                      CL[1]^2
                                     Wburn[1]   CD[1]     (1.01)
                                      (0.273lbf)  (0.0189) 1/Re[1]^0.2
                                                            (0.0772)

                                           CL[0]^2
                         Wburn[0]   CD[0]    (1.01)
                          (0.274lbf)  (0.019) 1/Re[0]^0.2
                                               (0.0772)
```

(continues on next page)

(continued from previous page)

```
               FlightSegment AircraftP

       Mission


 Model         Wfuel[0]  Wfuel[1] + Wburn[0]

                Wfuel[1]  Wfuel[2] + Wburn[1]
                Wfuel[2]  Wfuel[3] + Wburn[2]

                 Wing

       Aircraft W  Fuselage.W + Wing.W

                 Fuselage W = 100lbf



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

      | Aircraft.Wing
    S : 44.14                                       [ft^2] 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

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
    Wing.W  S·Wing.rho
+ Wburn[:]  0.2·D[:]

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

Relative Differences |above 1%|
-------------------------------
Wburn : [ +102.1%   +101.6%   +101.1%   +100.5%  ] segment fuel burn
```

(continues on next page)

(continued from previous page)

```
Wfuel : [ +101.3%   +101.1%   +100.8%   +100.5%  ] fuel weight
    D : [   +1.1%       -         -         -    ] drag force
```

CHAPTER 7

Advanced Commands

7.1 Choice Variables

If MOSEK 9 is installed, GPkit supports discretized free variables with the `mosek_conif` solver. Choice variables are free in the sense of having multiple possible choices, but discretized in the sense of having a limited set of possible solutions.

```
"Example choice variable usage"
import numpy as np
from gpkit import Variable, Model

x = Variable("x", choices=range(1, 4))
num = Variable("numerator", np.linspace(0.5, 7, 11))

m = Model(x + num/x)
sol = m.solve(verbosity=0)

print(sol.table())
```

If solved with the mosek_conif solver, the script above will print:

```
Optimal Cost
------------
 [ 1.5       2.15      2.8       3.22      ... ]

~~~~~~~~
WARNINGS
~~~~~~~~
No Dual Solution
----------------
This model has the discretized choice variables [x] and hence no dual
solution. You can fix those variables to their optimal value and get
sensitivities to the resulting continuous problem by updating your model's
substitions with `sol["choicevariables"]`.
```

(continues on next page)

(continued from previous page)

```
~~~~~~~~

Swept Variables
---------------
numerator : [ 0.5
              1.15
              1.8
              2.45
              3.1
              3.75
              4.4
              5.05
              5.7
              6.35
              7         ]

Free Variables
--------------
x : [ 1
      1
      1
      2
      2
      2
      2
      2
      2
      3
      3         ]
```

Note that the optimal values for `x` are discretized, clicking from 1 to 2 to 3 during the sweep, and that the solution has no dual variables.

7.2 Derived Variables

7.2.1 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, units

# 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: 1*units.day - 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)
```

(continues on next page)

(continued from previous page)

```
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.

7.2.2 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. When constructing an `evalfn`, remember that square-bracket access to variables pulls out magnitudes: use round-bracket access (i.e. `v(var)`) to ensure unit correctness.

```
"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
```

Note that this variable should not be used in constructing your model! For evaluated variables that can be used during a solution, see *Sequential Geometric Programs*.

7.3 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.

7.3.1 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.

7.3.2 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_cli": 1e-6, "mosek_conif": 1e-6,
        "cvxopt": 1e-7}[gpkit.settings["default_solver"]]
```

(continues on next page)

(continued from previous page)

```
# 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)`.

7.4 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
```

7.5 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
```

7.6 Substitutions

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

7.6.1 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
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.

7.6.2 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.

7.6.3 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).

7.6.4 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.

7.6.5 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`.

7.6.6 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"]
```

CHAPTER 8

Signomial Programming

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

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

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))

# use x0 to give the solution, reducing number of GPs needed
m.localsolve(verbosity=0, x0={x: 0.9, y:0.1})
assert len(m.program.results) == 2
```

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_0, however, you may do so by putting that position (a dictionary formatted as you would a substitution) as the `x0` 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,
              ]
```

(continues on next page)

(continued from previous page)

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

```
    Cost
 (0.785) 0.785




       x  0.785
 Model


       y  x


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

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):
```

(continues on next page)

(continued from previous page)

```
        "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 posynomial approximation around that position
        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())
```

```
    Cost
 (0.707) 0.785



       <external_constraint.ExternalConstraint object>
 Model


       x  0.785



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

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

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

CHAPTER 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

```
               /
         Cost
 (0.00367/m^3) /alpha
               / (2, fixed)




       A_{wall} = 200m^2
```

(continues on next page)

(continued from previous page)

```
 Model

       A_{wall}  2·h·w + 2·h·d


       alpha = 2

       alpha  h/w



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

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

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")
```

(continues on next page)

(continued from previous page)

```
# 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 [m^3]' to 'M [kg]'



                    d[0]     M
     Cost           (0.464m)  (100kg, fixed)
 (1.29m^2) A
           (1.29m^2)
                    d[1]·d[2]
                     (0.215m^2)




       A  2·(d[0]·d[1] + d[0]·d[2] + d[1]·d[2])


       M = 100kg


 Model
       M = V·\rho


       V = d[0]·d[1]·d[2]


       \rho = 1,000kg/m^3



Free Variables
--------------
A : 1.293                              [m^2] Surface Area of the Tank
V : 0.1                                [m^3] Volume of the Tank
d : [ 0.464     0.464     0.464    ]   [m]  Dimension Vector
```

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")
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,
```

(continues on next page)

(continued from previous page)

```
                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
======



                  W_0
   Cost           (4,940N, fixed)
 (303N) W
         (7,341N)
                  W_w      S
                   (2,401N)   (16.4m^2)




       W  W_0 + W_w

       C_D  (CDA0)/S + k·C_f·(\frac{S}{S_{wet}}) + C_L^2/(π·A·e)

       D  0.5·\rho·S·C_D·V^2

       W_0 = 4,940N

 ModelW  0.5·\rho·S·C_L·V^2

       e = 0.95
       (\frac{S}{S_{wet}}) = 2.05
       C_f  0.074/Re^0.2
       k = 1.2
```

(continues on next page)

(continued from previous page)

```
       [12 terms]



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

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
 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     ]  [m^2] 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

Solution Diff
```

(continues on next page)

(continued from previous page)

```
=============
(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 0.1|
--------------------------------
     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 ]  [m^2]  total wing area
V_{min} : [       -2        -2        +3        +3 ]  [m/s]  takeoff speed
      A : [     -2.3      -3.7     +0.38      -1.3 ]         aspect ratio
    C_L : [     -0.2      -0.3         -     -0.19 ]         Lift coefficent␣
↪of wing

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

    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,
```

(continues on next page)

(continued from previous page)

```
                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

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:

```
                                          th[2]
                                   w[2]   (0.285)
                                    (0.384m) th[1]
                            w[3]            w[1]
                             (0.76m) th[3]  th[2]
                                      (0.341)
                      w[4]          th[2]
                       (1.18m)
    Cost                     th[3] th[2]
 (1.62m) w[5]
          (1.62m)
```

(continues on next page)

(continued from previous page)

```
                          th[4]   th[2]
                           (0.363)

                  th[5]   th[2]
                   (0.367)

                  th[4] th[2]









             L = 5·dx



             L = 6m

             EI = 11,000N·m²
 Model      w[5]  w[4] + 0.5·dx·(th[5] + th[4])
       Beam th[2]  th[1] + 0.5·dx·(M[2] + M[1])/EI
             w[4]  w[3] + 0.5·dx·(th[4] + th[3])
             M[1]  M[2] + 0.5·dx·(V[1] + V[2])
             th[3]  th[2] + 0.5·dx·(M[3] + M[2])/EI
             V[3]  V[4] + 0.5·dx·(q[3] + q[4])
             th[1]  th[0] + 0.5·dx·(M[1] + M[0])/EI


             [17 terms]



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
```

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

gpkit.constraints.bounded module

gpkit.constraints.costed module

gpkit.constraints.gp module

gpkit.constraints.loose module

gpkit.constraints.model module

gpkit.constraints.prog_factories module

gpkit.constraints.relax module

Module contents

10.1.2 Submodules

10.1.3 gpkit.build module

10.1.4 gpkit.exceptions module

10.1.5 gpkit.globals module

10.1.6 gpkit.keydict module

10.1.7 gpkit.repr_conventions module

10.1.8 gpkit.small_classes module

10.1.9 gpkit.small_scripts module

10.1.10 gpkit.solution_array module

10.1.11 gpkit.units module

10.1.12 gpkit.varkey module

10.1.13 Module contents

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