# robust Documentation

***Release 0.0.0***

**Ali Saab and Berk Ozturk**

**Jul 01, 2020**

# Contents


**1 Robust optimization 101** **3**
- 1.1 Basic mathematical principles . . . 3

**2 Installing robust** **5**
- 2.1 Dependencies . . . 5
- 2.2 Clone + install **robust** . . . 5

**3 Getting started** **7**

**4 More advanced commands** **9**
- 4.1 Choosing between RGP approximation methods and uncertainty sets . . . 9
- 4.2 Simulating robust designs . . . 10

**5 Why robust optimization?** **13**
- 5.1 Comparison of general SO methods with RO . . . 13

**6 Mathematical moves for robust GPs/SPs** **15**
- 6.1 Linear programs (LPs) have tractable robust counterparts. . . . 15
- 6.2 Two-term posynomials are LP-approximable. . . . 16
- 6.3 All posynomials are LP-approximable. . . . 16
- 6.4 GPs have robust formulations. . . . 17
- 6.5 RSPs can be represented as sequential RGPs. . . . 17

**7 Approximations for tractable robust GPs** **19**
- 7.1 Simple Conservative Approximation . . . 19
- 7.2 Linearized Perturbations . . . 19
- 7.3 Best Pairs . . . 20

**8 Approaches to solving robust SPs** **21**
- 8.1 General RSP Solver . . . 21
- 8.2 Best Pairs RSP Solver . . . 22
- 8.3 Linearized Perturbations RSP Solver . . . 22

**9 Goal programming** **23**
- 9.1 Implementation . . . 23

**10 References** **25**

11 Indices and tables 27

**robust** is a framework for engineering system optimization under uncertainty using geometric and signomial programming.

Robust optimization is a tractable stochastic optimization method that protects against uncertain parameters in well-defined sets, and optimizes for the worst case objective.

This website is under construction. If you are not able to find answers to your questions in the documentation. please feel free to post issues and suggest areas for improvement.

Table of contents:

# CHAPTER 1

# Robust optimization 101

RO is a tractable method for optimization under uncertainty, and specifically under uncertain parameters. It optimizes the worst-case objective outcome over uncertainty sets, unlike general stochastic optimization methods which optimize statistics of the distribution of the objective over probability distributions of uncertain parameters. As such, RO sacrifices generality for tractability, probabilistic guarantees and engineering intuition.

## 1.1 Basic mathematical principles

[*paraphrased from Ozturk and Saab, 2019*]

Given a general optimization problem under parametric uncertainty, we define the set of possible realizations of uncertain vector of parameters $u$ in the uncertainty set $\mathcal{U}$. This allows us to define the problem under uncertainty below.

$$\begin{aligned} \min \;\; & f_0(x) \\ \text{s.t.} \;\; & f_i(x,u) \leq 0, \; \forall u \in \mathcal{U}, \; i = 1, \ldots, n \end{aligned}$$

This problem is infinite-dimensional, since it is possible to formulate an infinite number of constraints with the countably infinite number of possible realizations of $u \in \mathcal{U}$. To circumvent this issue, we can define the following robust formulation of the uncertain problem below.

$$\begin{aligned} \min \;\; & f_0(x) \\ \text{s.t.} \;\; & \max_{u \in \mathcal{U}} f_i(x,u) \leq 0, \; i = 1, \ldots, n \end{aligned}$$

This formulation hedges against the worst-case realization of the uncertainty in the defined uncertainty set. The set is often described by a norm, which contains possible uncertain outcomes from distributions with bounded support

$$\begin{aligned} \min \;\; & f_0(x) \\ \text{s.t.} \;\; & \max_{u} f_i(x,u) \leq 0, \; i = 1, \ldots, n \\ & \|u\| \leq \Gamma \end{aligned}$$

where $\Gamma$ is defined by the user as a global uncertainty bound. The larger the $\Gamma$, the greater the size of the uncertainty set that is protected against!

CHAPTER 2

# Installing **robust**

## 2.1 Dependencies

To be able to use **robust**, you will need the following software installed on your system:

- Python 3.0 or higher
- GPkit
- numpy
- scipy

Please click on each link to see installation instructions.

## 2.2 Clone + install robust

To install **robust**, clone convexengineering/robust into your directory of choice.

Using a console, go into the directory one level above the repository, and type in the following command:

```
pip install robust -e
```

Your local version of **robust** will be ready to go!

# CHAPTER 3

## Getting started

Once you have installed **robust**, and have a GP or SP model, you are ready to begin. From here onward, we will use *nominal* to describe models with no uncertainty straight out of GPkit, and *robust* to describe models that have been robustified using **robust**.

The uncertainties in **robust** are defined by adding attribute *pr* to any variable in your model. This attribute describes the $3\sigma$ uncertainty for the given parameter, normalized by its mean (otherwise known as 3 times the coefficient of variation). Note that these attributes are carried by nominal models but only come into effect when **robust** is applied.

```
from gpkit import Variable, Model
x = Variable('x', pr = 12) # 3CV = 12%
# ...
# after more variables, constraints
# ...
m = Model(objective, constraints, substitutions)
```

Once you have added uncertainties to parameters, and created a GPkit model, robustifying said model and solving it is easy. The most straight-forward inputs for uncertainty_set are 'box' or 'elliptical'. *gamma* defines the size of the uncertainty set protected against, where *gamma=1* protects against $3\sigma$ uncertainty.

```
from robust.robust import RobustModel
rm = RobustModel(m, uncertainty_set, gamma = float)
rsol = rm.robustsolve()
```

You have solved your robust model! To be able to quickly compare the robust solution *rsol* with the nominal solution *sol*, we recommend you try 'diffing' the two, which is done as follows:

```
print rsol.diff(sol)
```

This will allow you to see the percent differences between the two designs! Since the robust design protects against uncertainty in the parameters, it will necessarily have lower performance than the nominal design. If this has piqued your interest, please continue to explore the documentation.

# CHAPTER 4

# More advanced commands

*[Documentation work in progress...]*

**robust** has a variety of tools beyond the basics to aid engineering design under uncertainty.

All of these methods have been implemented in the robustSPpaper code repository, which defines the models used in [Ozturk, 2019].

## 4.1 Choosing between RGP approximation methods and uncertainty sets

There are many possible `**options` to `RobustModel`, but the primary function of the options is to be able to choose between the different RGP approximation methods defined in [Saab, 2018], which have differing levels of conservativeness. We prefer to define the three methods in a dict for ease access, and call `RobustModel`, for example for Best Pairs, as follows:

```
methods = [{'name': 'Best Pairs', 'twoTerm': True, 'boyd': False, 'simpleModel':␣
↪False},
           {'name': 'Linearized Perturbations', 'twoTerm': False, 'boyd': False,
↪'simpleModel': False},
           {'name': 'Simple Conservative', 'twoTerm': False, 'boyd': False,
↪'simpleModel': True}]
method = methods[0] # Best Pairs
robust_model = RobustModel(m, uncertainty_set, gamma=Gamma, twoTerm=method['twoTerm'],
                               boyd=method['boyd'], simpleModel=method['simpleModel'])
```

For `uncertainty_set`, `'box'` and `'elliptical'` are currently supported, and define the $\infty$- and 2-norms respectively.

## 4.2 Simulating robust designs

Robust optimization provides guarantees of constraint satisfaction under the defined uncertainty sets, but it is possible that the real distributions of uncertainty are better defined by probability distributions. Within `robust`, we have a framework to generate Monte Carlo (MC) simulations of the uncertain parameters to estimate the probability of constraint violation (i.e. probability of failure [pof]) of models, as well as the expectation and standard deviation of the objective cost. To generate samples of uncertain parameters and simulate a robust model's performance over these samples, we use the `simulate` module of `robust`. Good implementations of `robust.simulate` are in robustSPpaper/SimPleAC_pof_simulate.

There is one additional step before GPkit models can be simulated. Every free variable that is *fixed* during MC simulation must have a `fix = True` attribute, as following:

```
A          = Variable("A", "-", "aspect ratio", fix = True)
```

This allows for the optimizer to know that these variables (eg. aspect ratio in aircraft design), once the optimization is complete, cannot change under simulation. Other variables, eg. state variables such as speed and altitude of an aircraft, can change during MC simulation to fulfill constraints.

There are a daunting number of parameters for the functions in this module, most of which have little or no effect to optimization and simulation outcomes, but are necessary for the mathematics of RGPs. This I would recommend overcoming by creating a parameter function such as the following, which I have filled with some defaults:

```
def pof_parameters():
    model = SomeGPModelWithUncertainParams() # A GP model with uncertain parameters
    number_of_time_average_solves = 3 # number of solves for computing time cost, set␣
↪to 1 if not important

    ## PARAMETERS THAT AFFECT NUMBER OF GP MODELS SAMPLED
    # Each combination of methods, uncertainty_sets and gammas generates a
    # new RGP model designed with these inputs, to be tested against MC samples.
    methods = [{'name': 'Best Pairs', 'twoTerm': True, 'boyd': False, 'simpleModel':␣
↪False},
               {'name': 'Linearized Perturbations', 'twoTerm': False, 'boyd': False,
↪'simpleModel': False},
               {'name': 'Simple Conservative', 'twoTerm': False, 'boyd': False,
↪'simpleModel': True}
               ]
    uncertainty_sets = ['box', 'elliptical']
    nGammas = 11
    gammas = np.linspace(0, 1.0, nGammas)

    # LINEARIZATION PARAMETERS
    # These exist because RGPs/RSPs implement robust linear programs in their backend.
    # Defaults are great for engineering purposes.
    min_num_of_linear_sections = 3
    max_num_of_linear_sections = 99
    linearization_tolerance = 1e-4

    # PARALLELIZATION (will be available in future, please keep as False for now)
    parallel = False

    # SAMPLING PARAMETERS
    number_of_iterations = 100 # number of MC samples
    distribution = 'normal' # or 'uniform'

    # GENERATING SAMPLES (contained in directly_uncertain_vars_subs)
```

(continues on next page)

(continued from previous page)

```
    nominal_solution, nominal_solve_time, nominal_number_of_constraints, directly_
↪uncertain_vars_subs = \
        simulate.generate_model_properties(model, number_of_time_average_solves,␣
↪number_of_iterations, distribution)

    verbosity = 1 # controls printouts for simulations

    return [model, methods, gammas, number_of_iterations,
    min_num_of_linear_sections, max_num_of_linear_sections, verbosity, linearization_
↪tolerance,
    number_of_time_average_solves, uncertainty_sets, nominal_solution, directly_
↪uncertain_vars_subs, parallel,
            nominal_number_of_constraints, nominal_solve_time]
```

Simulating a GPkit model is equivalent to optimizing the model over its remaining free variables (without the 'fix' attribute). For Monte Carlo simulations, it is important to note that **the solution time is proportional to the product of number of methods, uncertainty sets, gammas and samples**. As such, one MC simulation usually takes similar to slightly less time to one solution to the un-robustified model. Furthermore, MC simulations take *longer* for robustified models vs. unrobust ones, since infeasibility can be detected faster than a feasible solution. If in a time crunch, it is recommended that one method, set and gamma is chosen for simulation purposes.

Once the parameters are generated, the `robust.simulate` module can be used to generate MC data.

```
solutions, solve_times, simulation_results, number_of_constraints = simulate.variable_
↪gamma_results(
                                        model, methods, gammas, number_of_iterations,
                                        min_num_of_linear_sections,
                                        max_num_of_linear_sections, verbosity,␣
↪linearization_tolerance,
                                        number_of_time_average_solves,
                                        uncertainty_sets, nominal_solution, directly_
↪uncertain_vars_subs, parallel=parallel)
```

It is highly recommended that you save/pickle the results, since MC simulations can have a large time cost. robustSP-paper/SimPleAC_save and other files in the `simulate` module have simple demonstrations of saving/pickling.

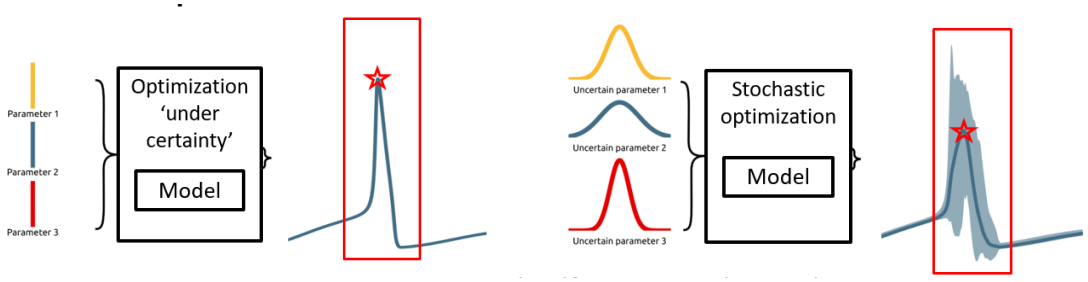
CHAPTER 5

# Why robust optimization?

Firstly, why optimization under uncertainty? Simply put, we want to preserve constraint feasibility under perturbations of uncertain parameters, with as little a penalty as possible to objective performance. In other words, we want designs that protect against uncertainty *least conservatively*, especially when compared with designs that leverage conventional design methods such as design with margins or multimission design.

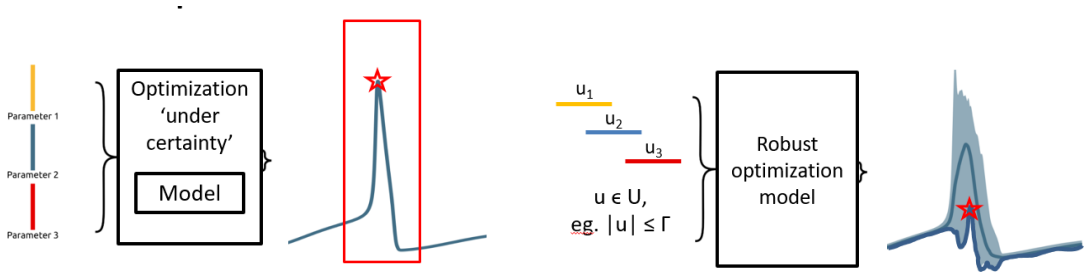
RO introduces mathematical rigor to design under uncertainty, and aims to reduce the sensitivity of design performance to uncertain parameters, thereby reducing risk.

## 5.1 Comparison of general SO methods with RO




General optimization 'under certainty', eg. gradient descent, is done using methods that sample the objective function, and use local information to converge towards an optimal solution. Stochastic optimization uses the same principles, but with the addition of uncertain parameters sampled from distributions. It then optimizes for some characteristic of the distribution of the objective, such as some risk measure or expectation.

Stochastic optimization has many benefits. It makes best use of available data about parameters, and it is extremely general. However, design outcomes can be significantly affected by the ability to sample from the parameter distribution, which in many cases is not well known. An even worse prospect for SO is the combinatorics and computational cost of probability distribution function (PDF) propagation through problem physics. The propagation of probability distributions of parameters requires the integration of PDFs with objective and constraint outcomes. Since this is difficult, this is often achieved through high-dimensional quadrature and the enumeration of potential outcomes into scenarios. And even so, SO has big computational requirements.




RO takes a different approach, choosing to optimize designs for worst-case objective outcomes over well-defined uncertainty sets. RO takes advantage of mathematical structure, requiring that the design problem is formulated as a program that has a tractable robust counterpart, such as an LP, QP, SDP, GP or SP. This is restrictive, but many engineering problems of interest can be formulated in these forms, with some significant benefits over general SO.

Within RO, the problem is monolithic; there is sampling from probability distributions, no separate evaluation step and optimization loop. RO problems are deterministic, with probabilistic guarantees of feasibility, and solve orders of magnitude faster than SO formulations with the same constraints. Furthermore, only the mild assumption of bounded uncertainty sets is required; no problem-specific approximations, assumptions or algorithms are needed. Any feasible GP or SP can be solved as an RO problem. As such, RO is especially suited to problems that are data deprived, such as conceptual design problems.

# CHAPTER 6

# Mathematical moves for robust GPs/SPs

There are 5 mathematical steps to being able to apply principles from linear robust optimization to geometric and signomial programming.

- Linear programs (LPs) have tractable robust counterparts.
- Two-term posynomials are LP-approximable.
- All posynomials are LP-approximable.
- GPs have robust formulations.
- RSPs can be represented as sequential RGPs.

As a quick demonstration, we paraphrase the foundational works of mathematics, in order of application, that allow us to come up with robust counterparts for GPs and SPs.

## 6.1 Linear programs (LPs) have tractable robust counterparts.

This is a seminal finding by [Ben-Tal, 1999] that derives the robust counterpart for a linear program given different types of bounded uncertainty sets. One of the problems they detail is below, which is a linear program in which the coefficients $\mathbf{a}_i$ are subject to an affine perturbation by uncertain parameters $u_i$, which are contained in an ellipsoidal uncertainty set.

$$\begin{aligned}
\min\ & \mathbf{c}^T\mathbf{x} \\
\text{s.t.}\ & \mathbf{a}_i\mathbf{x} \leq b_i,\ \forall \mathbf{a}_i \in \mathcal{U}_i,\ i = 1, \ldots, m, \\
& \mathcal{U} = \{(\mathbf{a}_1, \ldots, \mathbf{a}_m) : \mathbf{a}_i = \mathbf{a}_i^0 + \Delta_i u_i,\ i = 1, \ldots, m, \quad \|u\|_2 \leq \rho\}
\end{aligned}$$

The robust counterpart for the above linear program is given by the following:

$$\begin{aligned}
\min\ & \mathbf{c}^T\mathbf{x} \\
\text{s.t.}\ & \mathbf{a}_i^0 \leq b_i - \rho \left\|\Delta_i\mathbf{x}\right\|_2,\ \forall i = 1, \ldots, m,
\end{aligned}$$

This is tractable second-order cone program.

## 6.2 Two-term posynomials are LP-approximable.

We were not the first people interested in robust GPs formulations. Folks from Stephen Boyd's group at Stanford took the first concrete steps to combine principles of robust linear programming with GPs. Work described in [Hsiung, 2008] paves the way for low-error piecewise-linear (PWL) approximations of posynomials.

**Corollary 1** *For $r \geq 3$, the unique best $r$-term PWL convex lower approximation $\underline{h}_r\colon \mathbf{R}^2 \to \mathbf{R}$ of the two-term log-sum-exp function is*

$$\underline{h}_r(y_1, y_2) = \max\{y_1, \underline{a}^\star_{r-2}y_1 + \underline{a}^\star_1 y_2 + \underline{b}^\star_1, \underline{a}^\star_{r-3}y_1 + \underline{a}^\star_2 y_2 + \underline{b}^\star_2, \ldots, \underline{a}^\star_1 y_1 + \underline{a}^\star_{r-2}y_2 + \underline{b}^\star_{r-2}, y_2\} \tag{24}$$

*and the unique best $r$-term PWL convex upper approximation $\overline{h}_r : \mathbf{R}^2 \to \mathbf{R}$ is*

$$\overline{h}_r(y_1, y_2) = \underline{h}_r(y_1, y_2) + \underline{\epsilon}_\phi(r), \tag{25}$$

*where $a^\star_i, b^\star_i, i = 1, \ldots, r-2$ are the coefficients of the segments of $\underline{\phi}_r$ defined in (23).*

For derivation of robust GPs, the central finding is in Corollary 1 of the paper, which asserts that there is an analytical solution for the lowest-error lower and upper approximation of a two-term posynomial in log-space. An image of the corollary of the paper is above; the proof can be found in the paper.

## 6.3 All posynomials are LP-approximable.

We use the PWL two-term posynomial approximation above to approximate any posynomial with PWL approximations of two-term posynomials. The full recipe is described by [Saab, 2018]; here we demonstrate with a simple example from the paper. The following problem

$$\begin{aligned} \min\ & f \\ \text{s.t. } & \max\{M_1 + M_2 + M_3 + M_4\} \\ \leq 1 & \\ & \max\{M_5 + M_6\} \\ \leq 1 & \end{aligned}$$

is equivalent to

$$\begin{aligned} \min\ & f \\ \text{s.t. } & \max\{M_1 + e^{t_1}\} \\ \leq 1 & \\ & \max\{M_2 + e^{t_1}\} \\ \leq e^{t_1} & \\ & \max\{M_3 + M_4\} \\ \leq e^{t_2} & \\ & \max\{M_5 + M_6\} \\ \leq 1 & \end{aligned}$$

by adding auxiliary variables and using properties of inequalities.

## 6.4 GPs have robust formulations.

Since we can represent all posynomials as PWL functions, we can robustify GP inequalities. Note that equalities cannot be robustified like inequalities, since under perturbation they would be infeasible. As such, it is preferred that equalities are relaxed whenever possible in GP models that will be robustified.

The final addition to this framework to enable robust GPs is to separate posynomials according to the dependence of monomial terms, as described in [Saab, 2018] and [Ozturk, 2019]. We show an example of such a partition, borrowed from Ozturk et al..

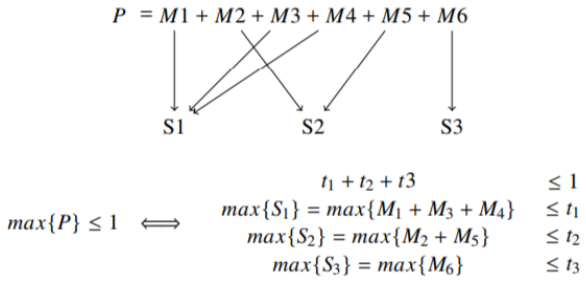



**Fig. 2 Partitioning of a large posynomial into smaller posynomials requires the addition of auxiliary variables. $S_i$ are posynomials with independent sets of variables.**

## 6.5 RSPs can be represented as sequential RGPs.

Just as SPs are solved as a sequence of GPs, RSPs can be solved as a sequence of RGPs. We solve the nominal RSP, then use the solution as the initial guess for the RSP solution heuristic outlined below.

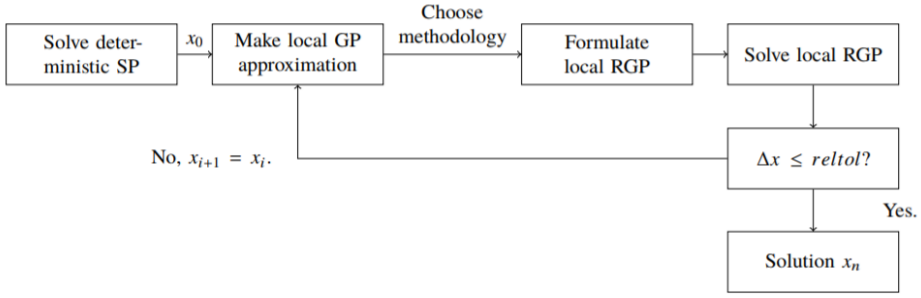



**Fig. 3 A block diagram showing the steps of solving a RSP.**

Within this framework, all GPs and SPs, given that they are feasible, have tractable robust counterparts.

# CHAPTER 7

# Approximations for tractable robust GPs

Within **robust**, there are 3 tractable approximate robust formulations for GPs, which can then be extended to SPs through heuristics The methods are detailed at a high level below, in decreasing order of conservativeness. Please see [Saab, 2018] for further details.

*(paraphrased from [Ozturk, 2019])*

The robust counterpart of an uncertain geometric program is:

$$\begin{aligned} \min \;\; & f_0(\mathbf{x}) \\ \text{s.t.} \;\; & \max_{\zeta \in \mathcal{Z}} \left\{ \sum_{k=1}^{K_i} e^{\mathbf{a_{ik}}(\zeta)\mathbf{x} + b_{ik}(\zeta)} \right\} \leq 1, \; \forall i \in 1, ..., m \end{aligned}$$

which is Co-NP hard in its natural posynomial form [Chassein, 2014]. We will present three approximate formulations of a RGP.

## 7.1 Simple Conservative Approximation

One way to approach the intractability of the above problem is to replace each constraint by a tractable approximation. Replacing the max-of-sum by the sum-of-max will lead to the following formulation.

$$\begin{aligned} \min \;\; & f_0(\mathbf{x}) \\ \text{s.t.} \;\; & \sum_{k=1}^{K_i} \max_{\zeta \in \mathcal{Z}} \left\{ e^{\mathbf{a_{ik}}(\zeta)\mathbf{x} + b_{ik}(\zeta)} \right\} \leq 1, \; \forall i \in 1, ..., m \end{aligned}$$

Maximizing a monomial term is equivalent to maximizing an affine function, therefore the Simple Conservative approximation is tractable.

## 7.2 Linearized Perturbations

The Linearized Perturbations formulation separates large posynomials into decoupled posynomials, depending on the dependence of monomial terms. If the exponents are known and certain, then large posynomial constraints can be

approximated as signomial constraints. The exponential perturbations in each posynomial are linearized using a modified least squares method, and then the posynomial is robustified using techniques from robust linear programming. The resulting set of constraints is SP-compatible, therefore, a robust GP can be approximated as a SP.

## 7.3 Best Pairs

If the exponents of a posynomial are uncertain as well as the coefficients, then large posynomials can't be approximated as a SP, and further simplification is needed. This formulation allows for uncertain exponents, by maximizing each pair of monomials in each posynomial, while finding the best combination of monomials that gives the least conservative solution. [Saab, 2018] provides a descent algorithm to find locally optimal combinations of the monomials, and shows how the uncertain GP can be approximated as a GP for polyhedral uncertainty, and a conic optimization problem for elliptical uncertainty with uncertain exponents.

To reiterate, please refer to [Saab, 2018] for further details on robust GP approximations.

# CHAPTER 8

# Approaches to solving robust SPs

*(paraphrased from [Ozturk, 2019])*

**robust** implements a heuristic to solve a RSP based on our previous discussion on robust geometric programming.

## 8.1 General RSP Solver

A common heuristic algorithm to solve a SP is by sequentially solving local GP approximations. Similarly, our approach to solve a RSP is based on solving a sequence of local RGP approximations. In this heuristic, a good initial guess will lead to faster convergence and possibly a better solution. The deterministic solution of the uncertain SP is in general a good candidate $x_0$.

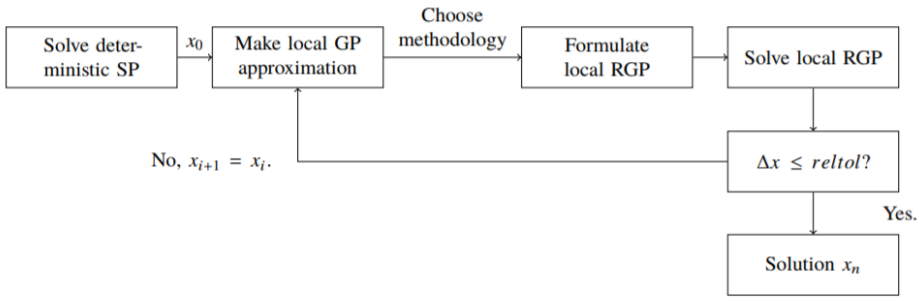



**Fig. 3 A block diagram showing the steps of solving a RSP.**

For comparisons between methods ahead, we write the algorithm explicitly as follows:

- Choose an initial guess $x_0$.
- Repeat:
  - Find the local GP approximation of the SP at $x_i$.
  - Find the RGP formulation of the GP.

  - Solve the RGP to obtain $x_{i+1}$.
  - If $x_{i+1} \approx x_i$: break

Any of the previously mentioned methodologies can be used to formulate the local RGP approximation. However, depending on the RGP formulation chosen to solve a RSP, the formulation and solution blocks in the above figure are adjusted.

## 8.2 Best Pairs RSP Solver

If the Best Pairs methodology is exploited, then the above algorithm would change so that each iteration would solve the local RGP approximation and choose the best permutation for each large posynomial. The modified algorithm would become as follows:

- Choose an initial guess $x_0$.
- Repeat:
  - Find the local GP approximation of the SP at $x_i$.
  - For each large posynomial constraint, select the new permutation $\phi$ such that $\phi$ minimizes the robust large constraint evaluated at $x_i$.
  - Solve the approximate tractable counterparts of the local GP, and let $\mathbf{x}_{i+1}$ be the solution.
  - If $x_{i+1} \approx x_i$: break.

## 8.3 Linearized Perturbations RSP Solver

On the other hand, if the Linearized Perturbations formulation is to be used, then we can avoid solving a SP at each iteration by first approximating the original SP constraints locally, and in the same loop approximating the robustified possibly signomial constraints locally, thus solving a GP at each iteration instead of a SP. The algorithm would then become as follows:

- Choose an initial guess $x_0$.
- Repeat:
  - Find the local GP approximation of the SP at $x_i$.
  - Robustify the constraints of the local GP approximation using the Linearized Perturbations methodology.
  - Find the local GP approximation of the resulting local SP at $x_i$.
  - Solve the local GP approximation in step c to obtain $x_{i+1}$.
  - If $x_{i+1} \approx x_i$: break.

Work in progress...

# CHAPTER 9

# Goal programming

Recall the standard RO form below

$$\begin{aligned} \min \;\; & f_0(x) \\ \text{s.t.} \;\; & \max_u f_i(x,u) \leq 0, \; i = 1, \ldots, n \\ & \|u\| \leq \Gamma \end{aligned}$$

where we attempt to minimize $f_0$ for the worst-case realization of uncertain parameters in the set. We can flip this on its head, and solve the following problem

$$\begin{aligned} \max \;\; & \Gamma \\ \text{s.t.} \;\; & f_i(x,u) \leq 0, \; i = 1, \ldots, n \\ & \|u\| \leq \Gamma \\ & f_0(x) \leq (1+\delta) f_0^*, \; \delta \geq 0 \end{aligned}$$

where $f_0^*$ is the optimum of the nominal problem and $\delta$ is a fractional penalty on the objective that we are willing to sacrifice for robustness, which gives $(1+\delta)f_0^*$ as the upper bound on the objective value.

The benefit of this goal programming form is that we can start to use risk as a global design variable against with all objectives can be weighed.

## 9.1 Implementation

To use the goal programming functions in **robust**, you can use the `simulate.variable_goal_results` function, which has the same inputs as `simulate.variable_gamma_results` except for having the penalty parameter $\delta$ instead of the uncertainty set size $\Gamma$ as its input.

What occurs to the solved nominal model under the hood is the following:

```
Gamma = Variable('\\Gamma', '-', 'Uncertainty bound')
delta = Variable(value, '1+\\delta', '-', 'Acceptable optimal solution bound', fix =␣
↪True)
```

(continues on next page)

(continued from previous page)

```
origcost = model.cost
mGoal = Model(1 / Gamma, [model, origcost <= Monomial(nominal_solution(origcost)) *␣
↪delta, Gamma <= 1e30, delta <= 1e30],
              model.substitutions)
robust_goal_model = RobustModel(mGoal, uncertainty_set, gamma=Gamma)
sol = robust_goal_model.robustsolve()
```

This is an exact formulation of the aforementioned risk minimization problem!

CHAPTER 10

# References


[Ben-Tal, 1999] Ben-Tal, A., and Nemirovski, A., "Robust solutions of uncertain linear programs," Operations Research Letters, 1999.

[Chassein, 2014] Chassein, A., and Goerigk, M., "Robust Geometric Programming is co-NP hard.", Fachbereich Mathematik, Technische Universitat Kaiserslautern, Germany, 2014, pp. 1–6.

[Hsiung, 2008] Hsiung, K. L., Kim, S. J., and Boyd, S., "Tractable approximate robust geometric programming," Optimization and Engineering, vol. 9, 2008, pp. 95–118.

[Ozturk, 2019] Ozturk, B. and Saab, A., "Optimal Aircraft Design Decisions Under Uncertainty via Robust Signomial Programming", AIAA Aviation 2019 Conference Proceedings.

[Saab, 2018] Saab, A., Burnell, E., and Hoburg, W., "Robust Designs via Geometric Programming", arXiv:1808.07192v1.

[Soyster, 1974] Soyster, A.L., "A Duality Theory for Convex Programming with Set-Inclusive Constraints", Operations Research, Vol. 22, No. 4, August 1974.


Work in progress...

CHAPTER 11

# Indices and tables

- genindex
- modindex
- search