Lecture 15: Multi-Objective Optimization in the IvP Helm

April 28th 2020

Outline

- What is an Optimization Problem?
- What is a Multi-Objective Optimization Problem?
- How is Multi-Objective Optimization Used on a Robot?
- Multi-Objective Optimization in the IvP Helm
- Introduction to the Tools for Creating Optimization Problems for the IvP Helm
A Function with a Single Optima

\[ f(x^*) = \arg \max_x f(x) \]

\[ x^* = 10 \]

\[ f(x^*) = 7 \]

\[ y = \frac{1}{\sigma \sqrt{2\pi}} e^{-\frac{(x - \mu)^2}{2\sigma^2}} \]

\( \mu = \text{Mean} \)

\( \sigma = \text{Standard Deviation} \)

\( \pi \approx 3.14159 \ldots \)

\( e \approx 2.71828 \ldots \)

A Function with a Multiple Optima

(“multi-modal” functions)
Discrete Optimization

Global Optima

Optimization
Problems
Multi-Objective
Optimization
Robot IvP Helm
Optimization
The IvP
Domain
IvP
Functions
IvPBuild
ZAIC Tools
IvPBuild
Reflector Tools

Michael Benjamin, Henrik Schmidt, ©2020
MIT Dept of Mechanical Engineering
Mathematical Programming

Real World Problem → Problem Format (instance) → Solution Algorithm → Fast Solutions (with guaranteed properties)

Optimization in Marine Autonomy

Consider the problem from Lab 14:
Two vehicles search for hazardous objects, while minimizing the reporting of false alarms.
Consider the problem from Lab 14: Two vehicles search for hazardous objects, while minimizing the reporting of false alarms.

Optimization is may be defined by the stated problem parameters:

Objective Function: \( \min c_1 x_1 + c_2 x_2 \)

Constraint: \( s.t. \ t < 9000 \)

where

\( t \) is the total mission time in seconds
\( x_1 \) is the number of missed hazards
\( x_2 \) is the number of false alarms
\( c_1 \) is 100 (the penalty for missing a hazard)
\( c_2 \) is 35 (the false alarm penalty, for claiming a benign object as a hazard)

Linear Programming

A simple example (from Ecker, Kupferschmid, 1988):

Objective Function: maximize \( z = 20x_1 + 15x_2 \)

subject to:

\[
\begin{align*}
2x_1 - x_2 & \leq 8 \\
x_2 & \leq 0 \\
2x_1 + x_2 & \leq 12.5 \\
x_1 & \geq 0 \\
x_2 & \geq 0
\end{align*}
\]

The Oakwood Furniture Company has 12.5 units of wood on hand from which to manufacture tables and chairs. Making a table uses two units of wood and making a chair uses one unit. Oakwood’s distributor will pay $20 for each table and $15 for each chair, but he will not accept more than eight chairs and he wants at least twice as many chairs as tables. How many tables and chairs should the company produce to maximize revenue?
Linear Programming
Feasible Space

The Oakwood Furniture Company Example (from Ecker, Kupferschmid, 1988)

\[
\begin{align*}
2x_1 + x_2 & \leq 12.5 & \text{(raw material)} \\
x_2 & \leq 8 & \text{(at most 8 chairs)} \\
2x_1 & \leq x_2 & \text{(at least 2 chairs for every table)} \\
x_1, x_2 & \geq 0 & \text{(non-negativity)} \\
\end{align*}
\]

Objective Function: \( z = 20x_1 + 15x_2 \)

Graphical solution method: Really only for illustration and works only in 2D. If you could draw a line representing the objective function and “slide” it down the page until it hits a vertex...
Graphical solution method: Really only for illustration and works only in 2D.

If you could draw a line representing the objective function and “slide” it down the page until it hits a vertex...

Objective Function: \[ z = 20x_1 + 15x_2 \]

Note the optimal solution of 2.25 tables and 8 chairs.
When the decision space is discrete, the problem may be better formulated as an integer programming problem.
These problems are harder to solve generally – why?

Objective Function: \[ z = 20x_1 + 15x_2 \]
Simplex Algorithm

Overview

- Capitalizes on the fact that the optimal solution resides at one of the vertices of the feasible space.
- The number of vertices in real-world problems is unmanageably large. It would not be practical to exhaustively investigate all vertices.
- Simplex proceeds from one vertex to another neighboring vertex, always improving on the solution. In practice it is very fast.
- Simplex first published in 1948.
- The journal Computing in Science and Engineering listed it as one of the top 10 algorithms of the twentieth century.
- That being said, Simplex has been improved upon, including a class of algorithms referred to as interior methods that migrate from vertex to vertex more efficiently.

The Key Idea of Mathematical Programming

1) The LP format has expressive power (Many real problems are LP problems)
2) The LP format may be exploited algorithmically (guaranteed fast solutions)
Mathematical Programming

All the World’s Problems

bigger subset

subset

Clever Transformation Methods

Problem Format (instances)

Solution Algorithm

Fast Solution (with guaranteed properties)

Problem Format

Solution Algorithm

What is the relevance of Linear Programming to Marine Autonomy?

Probably very little!
Consider the problem from Lab 14: Two vehicles search for hazardous objects, while minimizing the reporting of false alarms.

Optimization is may be defined by the stated problem parameters:

Objective Function: \[ \min C_1 x_1 + C_2 x_2 \]

Constraint: \[ s.t. \quad t < 9000 \]

where

- \( t \) is the total mission time in seconds
- \( x_1 \) is the number of missed hazards
- \( x_2 \) is the number of false alarms
- \( c_1 \) is 100 (the penalty for missing a hazard)
- \( c_2 \) is 35 (the false alarm penalty, for claiming a benign object as a hazard)

In marine robotic platforms, and real world robots generally, decision making happens at several levels:

- Where and when is the next destination?
- What is our path plan?
- What are the sequence of heading and speed commands?
- What are the sequence of rudder and thrust commands?
In marine robotic platforms, and real world robots generally, decision making happens at several levels:

- Where and when is the next destination?
- What is our path plan?
- What are the sequence of heading and speed commands?
- What are the sequence of rudder and thrust commands?

Question:
- How much of this can be sorted out before launch?
- How much is determined on-the-fly?

Optimization in Marine Autonomy

In marine robotic platforms, and real world robots generally, decision making happens at several levels:

- Where and when is the next destination?
- What is our path plan?
- What are the sequence of heading and speed commands?
- What are the sequence of rudder and thrust commands?

Mission Autonomy

Platform Autonomy

Platform Control

Vehicle Agnostic (mostly)

Vehicle Dependent (mostly)
If the World Were Totally Predictable (but it’s not)

**Fake World: Total Predictability**

**Real World: Some Predictability and a lot of improvising**

Time Labse SAIL 2015: https://www.youtube.com/watch?v=BFoCPF06kn

Optimization in Marine Autonomy

In the Linear Programming example, the decision was simply to decide how many chairs, and how many tables to build \((x_1, \text{ and } x_2)\).

In robotic platforms, decision making happens at several levels:

- Where and when is the next destination?
- What is our path plan?
- What are the sequence of heading and speed commands?
- What are the sequence of rudder and thrust commands?

**OUR FOCUS**

**Mission Autonomy**

**Vehicle Agnostic**
(mostly)

**Platform Autonomy**

**Vehicle Dependent**
(mostly)

**Platform Control**
Multi-Objective Optimization

Concept introduction

- Multiple objective functions over the same decision space.
- Metrics are typically uncorrelated – optimizing apples vs. oranges.
- Let’s return to the furniture example: It has single objective function – to maximize revenue:

Objective Function: maximize \[ z = 20x_1 + 15x_2 \]

subject to:

\[ x_2 \leq 8 \]
\[ 2x_1 - x_2 \leq 0 \]
\[ 2x_1 + x_2 \leq 12.5 \]
\[ x_1 \geq 0 \]
\[ x_2 \geq 0 \]

The Oakwood Furniture Company has 12.5 units of wood on hand from which to manufacture tables and chairs. Making a table uses two units of wood and making a chair uses one unit. Oakwood’s distributor will pay $20 for each table and $15 for each chair, but he will not accept more than eight chairs and he wants at least twice as many chairs as tables. How many tables and chairs should the company produce to maximize revenue?

Question: What if Oakwood also wants to maximize market presence? In other words, sell as many items as possible, chairs or tables.
Multi-Objective Optimization

Concept introduction

- Multiple objective functions over the same decision space.
- Metrics are typically uncorrelated – optimizing apples vs. oranges.

Objective Function:

maximize \( z = x_1 + x_2 \)

subject to:

\[
\begin{align*}
x_2 & \leq 8 \\
2x_1 - x_2 & \leq 0 \\
2x_1 + x_2 & \leq 12.5 \\
x_1 & \geq 0 \\
x_2 & \geq 0
\end{align*}
\]

The end-goal may be to maximize revenue “overall in the future”. The management may be certain that they want to both maximize profit and maximize market share, but may have no clue what the right mix may be to maximize revenue 10 years out.

Not knowing what that “right mix” may be wouldn’t preclude trying to at least explore what the options may be. (Exploring the Pareto frontier).

Multi-Objective Optimization

Definition

- A multi-objective optimization problems may be expressed as

\[
\min_x f_1(x), f_2(x), \ldots, f_n(x)
\]

Typically there is no definitive solution to this problem, but rather a family of solutions – Pareto Optimal solutions.

A Pareto optimal solution is one where improvement on one objective function cannot be achieved without sacrificing performance on another objective function.

A Pareto Optimal solution is also called a non-dominated solution.
### Pareto Optimality

**Simple Example**

Your goal after graduation is to find a job that both:
- Pays well
- Close to where your significant other lives.

<table>
<thead>
<tr>
<th>Company</th>
<th>Salary</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>iRobot</td>
<td>$65,000</td>
<td>37 miles</td>
</tr>
<tr>
<td>Bluefin Robotics</td>
<td>$86,000</td>
<td>55 miles</td>
</tr>
<tr>
<td>Clearpath Robotics</td>
<td>$102,000</td>
<td>342 miles</td>
</tr>
<tr>
<td>Rethink Robotics</td>
<td>$82,000</td>
<td>45 miles</td>
</tr>
<tr>
<td>Robotic Marine Systems</td>
<td>$47,000</td>
<td>65 miles</td>
</tr>
<tr>
<td>Jaybridge Robotics</td>
<td>$54,000</td>
<td>119 miles</td>
</tr>
<tr>
<td>Boston Dynamics</td>
<td>$92,000</td>
<td>76 miles</td>
</tr>
<tr>
<td>Black-I Robotics</td>
<td>$84,000</td>
<td>122 miles</td>
</tr>
<tr>
<td>Honeybee Robotics</td>
<td>$39,000</td>
<td>144 miles</td>
</tr>
<tr>
<td>Friendly Robotics</td>
<td>$69,000</td>
<td>94 miles</td>
</tr>
</tbody>
</table>

**Dominated choices**

Your goal after graduation is to find a job that both:
- Pays well
- Close to where your significant other lives.

<table>
<thead>
<tr>
<th>Company</th>
<th>Salary</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>iRobot</td>
<td>$65,000</td>
<td>37 miles</td>
</tr>
<tr>
<td>Bluefin Robotics</td>
<td>$86,000</td>
<td>55 miles</td>
</tr>
<tr>
<td>Clearpath Robotics</td>
<td>$102,000</td>
<td>342 miles</td>
</tr>
<tr>
<td>Rethink Robotics</td>
<td>$82,000</td>
<td>55 miles</td>
</tr>
<tr>
<td>Robotic Marine Systems</td>
<td>$47,000</td>
<td>65 miles</td>
</tr>
<tr>
<td>Jaybridge Robotics</td>
<td>$54,000</td>
<td>119 miles</td>
</tr>
<tr>
<td>Boston Dynamics</td>
<td>$92,000</td>
<td>76 miles</td>
</tr>
<tr>
<td>Black-I Robotics</td>
<td>$84,000</td>
<td>122 miles</td>
</tr>
<tr>
<td>Honeybee Robotics</td>
<td>$39,000</td>
<td>144 miles</td>
</tr>
<tr>
<td>Friendly Robotics</td>
<td>$69,000</td>
<td>94 miles</td>
</tr>
</tbody>
</table>
Multi-Objective Optimization

**Definition**

- A multi-objective optimization problems may be expressed as

\[
\min_{x} f_1(x), f_2(x), \ldots, f_n(x)
\]

The term *value function* is sometimes used to refer to the decision-makers relative preference in optimizing each objective function.

The user may not precisely know his own value function, but may come to discover it by exploring tradeoffs in the Pareto Optimal space. (A good visualization GUI helps.)

Solutions that perform well in multiple criteria are often implied to be Pareto Optimal.
Solutions that perform well in multiple criteria are often implied to be Pareto Optimal.

This car may not even be Pareto Optimal w.r.t. “Great ride” and “Gets great mileage”!!

Pareto Optimal solutions are conflated with being optimal across all value functions.
Pareto Optimal solutions are conflated with being optimal across all value functions

<table>
<thead>
<tr>
<th>Torque</th>
<th>Mileage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ford</td>
<td>398 ft-lbs</td>
</tr>
<tr>
<td>Other</td>
<td>380 ft-lbs</td>
</tr>
</tbody>
</table>

- Claims to be Pareto Optimal (and maybe it is!)
- It may just dominate all other trucks on the issue of “torque”, and be worst on gas mileage

What is the role of Multi-Objective Optimization on a Robot?

- The decision-maker is the robot.
- It has to have a clear relative preference between objective functions (value function)
- It needs to make a decision and move on!
Multi-Objective Optimization On a Robot

- The robot multi-objective optimization problems may be expressed as

\[ \min_x f_1(x), f_2(x), \ldots, f_n(x) \]

How does the robot automatically solve a multi-objective optimization problem. By automatically we mean, automatically determine the value function (relative importance of functions).

If there were a human in the loop, they could explore different value functions, e.g., trading off distance travelled, vs. safety etc.

But there is no human in the loop, and the decisions are happening several times per second. The robot must have a value function – relative weight of its multiple objective functions.

Methods for Setting a Value Function

There are a few different ways to automatically setting the value function in a multi-objective optimization problem.

- Pick the most important objective function and ignore the rest. Seems draconian but on a robot this may make sense in some applications.

- “Good enough” search. Define a level of performance for each objective function that is good enough. Optimization of one objective function is done over a set of decisions deemed to satisfy the good enough criteria of preceding objective functions.

- Constructing a single aggregate objective function. Each objective function is given a priority weight.
Multi-Objective Optimization in the IvP Helm

Objective Functions

Behavior 1 \( f_1(x_1, x_2, \ldots, x_n) \)
Behavior 2 \( f_2(x_1, x_2, \ldots, x_n) \)
Behavior 3 \( f_3(x_1, x_2, \ldots, x_n) \)

Solver

\( \vec{x}^* = \arg\max_{\vec{x}} \sum_{i=1}^{k-1} w_i f_i(\vec{x}) \)

Action

- The solution, \( \vec{x}^* \), is the single decision maximizing the weighted sum of all utility functions.

The IvPDomain

Overview

The \textit{IvPDomain} is a data structure representing the decision space common to all objective functions produced by helm behaviors:

Behavior 1 \( f_1(x_1, x_2, \ldots, x_n) \)
Behavior 2 \( f_2(x_1, x_2, \ldots, x_n) \)
Behavior 3 \( f_3(x_1, x_2, \ldots, x_n) \)

A single IvPDomain

Solver

\( \vec{x}^* = \arg\max_{\vec{x}} \sum_{i=1}^{k-1} w_i f_i(\vec{x}) \)

Action
The IvPDomain is a discrete domain.

- There are a finite amount of possible decisions.
- The possible decisions are uniformly spaced, e.g., \( \text{speed} = \{0, 0.5, 1.0, 1.5, 2.0\} \)
- Brute force (exhaustive enumeration) in theory is an option, in practice too slow.
- A solution algorithm that explicitly or implicitly considers all decisions may be considered globally optimal.

The domain is discrete since control over the vehicle actuators may only have limited precision.

- Fruitless to reason about a heading=45.0024 if the vehicle’s heading sensor only provides precision to the nearest degree.
- Even if the sensor were able to provide this information, there may be limited difference in utility between 45.1 and 45.2 degrees.
- Vehicle heading is reconsidered on each iteration. So even though a heading of 45.72 degrees may be needed to reach a waypoint a kilometer away, a series of heading commands alternating between 45 and 46 can achieve this goal.
The IvP Domain

Defining the domain in the mission file

The IvP Domain is defined in the pHelmIvP configuration block in the mission.moos configuration file:

```
ProcessConfig = pHelmIvP
{
    AppTick = 4
    CommTick = 4
    Behaviors = charlie.bhv
    Verbose = true
    Domain = course:0:359:360
    Domain = speed:0:4:21
    Domain = depth:0:490:491
}
```

The above domain has three decision variables, course, speed, and depth.

- The course variable has 360 choices ranging from 0 to 359 degrees.
- The speed variable has 21 choices ranging from 0 to 4 meters per sec.
- The depth variable has 491 choices ranging from 0 to 490 meters.

All helm behaviors must reason over one or all of these three variables.

---

Helm Domain Handling

Events upon helm start-up

1. Helm is launched.

2. IvP Domain read from configuration file.

```
ProcessConfig = pHelmIvP
{
    domain = course:0:359:360
    domain = speed:0:4:21
    domain = depth:0:490:491
}
```

3. Behaviors spawned by helm, passing IvPDomain to each upon instantiation.
The IvPDomain
Receiving and Refining in the Constructor

The Constructor:

```c++
// Behavior Constructor
02 BHV_SimpleWaypoint::BHV_SimpleWaypoint(IvPDomain domain) :
03 IvPBehavior(domain)
04 {
05 m_domain = subDomain(m_domain, "course,speed");
06 addInfoVars("NAV_X, NAV_Y");
07 }
```

- Lines 2-3
  - The `domain` is passed to the behavior upon instantiation by the helm.
  - The `domain` is known to the helm from the helm configuration block.
  - The `domain` is handled by the `IvPBehavior` superclass constructor.
  - The result is that the `m_domain` member variable reflects the `domain`.

- Line 5
  - The `domain` is refined down to include only `course` and `speed` variables.
  - This objective function produced by this behavior will be defined only over `course` and `speed`.

IvP Sub-Domains
Definition and example

An IvPDomain A is a sub-domain of another IvPDomain B, if
- the set of decision variables in A is a subset of the variables in B, and
- the set of decisions for each variable is the same for each domain.

```c++
ProcessConfig = pHelmIvP A
{  
  Domain = course:0:359:360
  Domain = speed:0:4:21
  Domain = depth:0:490:491
}
```

```c++
ProcessConfig = pHelmIvP B
{  
  Domain = course:0:359:360
  Domain = speed:0:4:21
  Domain = depth:0:490:491
}
```

Is B a sub-domain of A? Yes
IvP Sub-Domains
Definition and example

A IvPDomain A is a sub-domain of another IvPDomain B, if
• the set of decision variables in A is a subset of the variables in B, and
• the set of decisions for each variable is the same for each domain.

\[
\begin{align*}
\text{ProcessConfig} &= \text{pHelmIvP} \\
&= \{
\text{Domain} = \text{course}:0:359:360 \\
\text{Domain} &= \text{speed}:0:4:21 \\
\text{Domain} &= \text{depth}:0:490:491
\}
\end{align*}
\]

\[
\begin{align*}
\text{ProcessConfig} &= \text{pHelmIvP} \\
&= \{
\text{Domain} = \text{course}:0:359:360 \\
\text{Domain} &= \text{speed}:0:4:21
\}
\end{align*}
\]

Is B a sub-domain of A? Yes

IvP Sub-Domains
Definition and example

A IvPDomain A is a sub-domain of another IvPDomain B, if
• the set of decision variables in A is a subset of the variables in B, and
• the set of decisions for each variable is the same for each domain.

\[
\begin{align*}
\text{ProcessConfig} &= \text{pHelmIvP} \\
&= \{
\text{Domain} = \text{course}:0:359:360 \\
\text{Domain} &= \text{speed}:0:4:21 \\
\text{Domain} &= \text{depth}:0:490:491
\}
\end{align*}
\]

\[
\begin{align*}
\text{ProcessConfig} &= \text{pHelmIvP} \\
&= \{
\text{Domain} = \text{course}:0:359:360 \\
\text{Domain} &= \text{speed}:0:4:5
\}
\end{align*}
\]

Is B a sub-domain of A?
IvP Sub-Domains
Definition and example

A IvPDomin A is a sub-domain of another IvPDomin B, if
• the set of decision variables in A is a subset of the variables in B, and
• the set of decisions for each variable is the same for each domain.

ProcessConfig = pHelmIvP  
{  
  Domain = course:0:359:360  
  Domain = speed:0:4:21  
  Domain = depth:0:490:491  
}

ProcessConfig = pHelmIvP  
{  
  Domain = course:0:359:360  
  Domain = speed:0:4:5  
}

Is B a sub-domain of A?  NO

IvP Sub-Domains
The subDomain() Utility Function

The subDomain() utility function may be used to create a proper subdomain of
another given IvPDomin:

Defined in lib_ivpbuild/BuildUtils.h

```cpp
IvPDomin  subDomain(IvPDomin, string);
```

```cpp
01  // Behavior Constructor  
02  BHV_SimpleWaypoint::BHV_SimpleWaypoint(IvPDomin domain) :  
03     IvPBehavior(domain)  
04  {  
05     m_domain = subDomain(m_domain, "course,speed");  
06     addInfoVars("NAV_X, NAV_Y");  
07  }
```
IvP Sub-Domains
The subDomain() Utility Function

If the domain variables specified in the subDomain() call are not in the given domain, the returned domain will be empty.

```cpp
m_domain = subDomain(m_domain, "course,speed");
if(m_domain.size() == 0)
    return(false);
```

IvP Domains and IvP Functions

An IvP Function is defined over an IvP Domain.

What is an IvP Function?
An IvP function is a piecewise linear approximation of an objective function, over a discrete decision space (domain).

\[ f_i(x, y) = ((1 - \frac{\sqrt{(x - 250)^2 + (y - 250)^2}}{250} \cdot 200) - 100, + ((1 - \frac{\sqrt{(x - 50)^2 + (y - 50)^2}}{250} \cdot 200) - 100) \]

Piecewise Linear Approximation 525 Pieces

Piecewise Linear Approximation 100 Pieces
**IvP Functions**

The IvP Function vs. Underlying Function

An **IvP function** is a piecewise linear approximation of an objective function, over a discrete decision space (domain).

![IvP Function](image)

Underlying Function

$$f(x,y) = \frac{1}{250} \left( (1 - \sqrt{(x - 250)^2 + (y - 250)^2} - 100) x + 200 \right) - 100 + \frac{1}{250} \left( (1 - \sqrt{(x - 50)^2 + (y - 50)^2} - 100) x + 200 \right) - 100$$

**Piecewise Linear Approximation 100 Pieces**

**IvP Functions**

Piece Format and Properties

**Piecewise linear (IvP) functions:**

- Each point in the decision space is contained by exactly one piece
- Each pieces has an interval boundary and a linear interior function.

**Limitations:**

- A piecewise linear function is only an approximation of the underlying function.
- But - the user has discretion over the number of pieces, distribution of pieces and time used to create the approximation.

Interval Boundary:

- $10 \leq x \leq 20$
- $14 \leq y \leq 21$

Interior Function:

$$f(x,y) = 4x + 8y + 7$$
IvP Functions

Piece Intersection:

- In any one given IvP function, no two pieces intersect (overlap).
- In the IvP solution algorithm (over multiple functions) piece intersection is a key idea.

\[ P_a \cap P_b \]

Boundary:
\[ 10 \leq x \leq 20 \]
\[ 14 \leq y \leq 21 \]
Interior Function:
\[ f(x,y) = 4x + 8y + 7 \]

Boundary:
\[ 15 \leq x \leq 25 \]
\[ 21 \leq y \leq 30 \]
Interior Function:
\[ f(x,y) = 2x + 5y + 2 \]

Boundary:
\[ 15 \leq x \leq 20 \]
\[ 21 \leq y \leq 21 \]
Interior Function:
\[ f(x,y) = 6x + 13y + 9 \]

Piece Intersection:
- In any one given IvP function, no two pieces intersect (overlap).
- In the IvP solution algorithm (over multiple functions) piece intersection is a key idea.

IvP Functions and IvP Behaviors

The primary output of a IvPBehavior is an IvPFunction, in the onRunState() function.

```cpp
IvPFunction *BHV_YourBehavior::onRunState()
{
    IvPFunction *ipf = generateIvPFunction()
    return(ipf);
}
```

The IvPBuild Toolbox (lib_ivpbuild) contains a number of “build tools” to facilitate the production of IvP Functions.

We explore this next.
The IvP Build Toolbox

A set of utilities for building IvP functions

Mathematical Programming

All the World’s Problems
IvP Helm Behaviors

subset
bigger subset

Clever Transformation Methods

The IvPBuild Toolbox

Problem Format
(instances)

IvP Functions

Solution Algorithm

IvP Solver

Fast Solution
(with guaranteed properties)

HEADING, SPEED, DEPTH
Decisions to the MOOSDB

Problem Format

Solution Algorithm
The IvPBuild Toolbox

General Usage Pattern

The **IvPBuild Toolbox**: a set of tools to facilitate building IvPFunctions. The tools each do different things, but all work in the same general way:

1. Create a **BuildTool** instance.
2. Pass the **IvPDomain** to the tool.
3. Pass Parameters to the tool.
4. Extract the **IvPFunction**.

The **IvPBuild Toolbox**: a set of tools to facilitate building IvPFunctions. The tools each do different things, but all work in the same general way:

- **The ZAIC Toolset**
  - Building 1 dimension objective functions (functions over a single variable)

- **The Reflector Toolset**
  - Building N dimension objective functions (functions over a multiple variables)
The ZAIC build tools are a family of tools for generating objective functions with a single decision variable. There are four tools in this set:

1. ZAIC_PEAK
2. ZAIC_LEQ
3. ZAIC_HEQ
4. ZAIC_Vector

The ZAIC_Peak Tool

The ZAIC_Peak tool is designed with the objective function form below in mind.

- A preferred decision (the summit), with maximum utility (maxutil).
- A drop-off in utility as the variable value deviates from the preferred choice.
Optimization

Problems

Robot IvP Helm
Optimization

Scoping

MOOS

IvP

Functions

IvPBuild
ZAIC Tools

The IvP Domain

IvPBuild
Reflector Tools

Michael Benjamin, Henrik Schmidt, ©2020

MIT

Dept of Mechanical Engineering

Multi-Objective Optimization

65

The ZAIC Build Tools

The ZAIC_Peak Tool

Typical usage of the tool in code:

```
01 ZAIC_PEAk zaic_peak(domain, "depth");
02 zaic_peak.setSummit(150);
03 zaic_peak.setMinMaxUtil(20, 120);
04 zaic_peak.setBaseWidth(60);
05 IvPFunction *ipf = 0;
06 ipf = zaic_peak.extractIvPFunction();
```

01 Create the ZAIC instance, passing the overall IvPDomain and particular variable.

03-05 Set the desired ZAIC parameters.

07-08 Extracting the IvPFunction from the ZAIC tool.

The ZAIC_LEQ Tool

The ZAIC_LEQ tool is designed with the objective function form below in mind.

- The `summit` parameter is the point where max utility begins to drop off.
- The `minutil` parameter has default 0. The `maxutil` parameter has default 100.
- The `basewidth` parameter may be used to soften the drop in utility.
The ZAIC_Peak Tools

- When $\text{basewidth} = 0$

$$f(x) = \begin{cases} \maxutil & x \leq \text{summit}, \\ \minutil & \text{otherwise}. \end{cases}$$

- When $\text{basewidth} \neq 0$

$$f(x) = \begin{cases} \maxutil \minutil + (\maxutil - \minutil) \times (x - \text{summit}) / \text{basewidth}, \\ \minutil & \text{otherwise}. \end{cases}$$

The ZAIC_LEQ Tool

Typical code structure:

```
01 ZAIC_LEQ zaic(domain, "depth");
02 zaic.setSummit(150);
03 zaic.setBaseWidth(60);
05 IpVF *ipf = 0;
06 ipf = zaic.extractIvPFunction();
```

01 Create the ZAIC instance, the overall IvPDoman and particular variable.
03-04 Set the desired ZAIC parameters.
06-07 Extracting the IvPFunction from the ZAIC tool.
The ZAIC_HEQ Tool

The ZAIC_HEQ tool is designed with the objective function form below in mind.

- The \textit{summit} parameter is the point where max utility begins to drop off.
- The \textit{minutil} parameter has default 0. The \textit{maxutil} parameter has default 100.
- The \textit{basewidth} parameter may be used to soften the drop in utility.

![Decision variable domain]

Typical code structure:

```plaintext
01  ZAIC_HEQ zaic(domain, "depth");
02  zaic.setSummit(150);
03  zaic.setBaseWidth(60);
04  IvPFunction *ipf = 0;
05  ipf = zaic.extractIvPFunction();
```

01  Create the ZAIC instance, the overall IvPDoman and particular variable.
03-04  Set the desired ZAIC parameters.
06-07  Extracting the IvPFunction from the ZAIC tool.
The ZAIC_Vector Tool

The ZAIC_Vector Tool is a catch-all tool for one-variable objective functions

- It accepts two equally sized vectors of numerical values (doubles).
- A vector of domain values.
- A vector of utility values.

Typical code structure:

```
01 ZAIC_Vector zaic(domain, "depth");
02 vector<double> domain;
03 vector<double> range;
04 domain.push_back(100); range.push_back(80);
05 domain.push_back(150); range.push_back(15);
06 domain.push_back(180); range.push_back(35);
07 zaic.setDomainVals(domain);
08 zaic.setRangeVals(range);
09 IvPFunction *ipf = 0;
10 ipf = zaic.extractIvPFunction();
```

- Create the ZAIC instance, the overall IvP Domain and particular variable.
- Build the pair of vectors.
- Pass the vectors to the ZAIC tool.
- Extracting the IvPFunction from the ZAIC tool.
The OF_Coupler Tool

The OF_Coupler tool is used to combine two objective functions into a single, combined function.
- The two objective functions must be defined over different variables.
- The resulting function will be defined over the coupled domain.

```
IvPFunction *ipf_1 = zaic_1.extractIvPFunction();
IvPFunction *ipf_2 = zaic_2.extractIvPFunction();
OF_Coupler coupler;
IvPFunction *ipf = coupler.couple(ipf_1, ipf_2, 50, 50);
```

Typical usage of the coupler in code:

01 The first IvP Function is created.
02 The second IvP Function is created.
03 A OF_Coupler tool is created.
04 The new coupled IvP Function is generated.

Note: the weights (50, 50) reflect the relative contribution of each function to the coupled function.
The OF_Coupler Tool

Typical usage of the coupler in code:

```cpp
01 IvPFunction *ipf_1 = zaic_1.extractIvPFunction();
02 IvPFunction *ipf_2 = zaic_2.extractIvPFunction();
03
04 OF_Coupler coupler;
05 IvPFunction *ipf = coupler.couple(ipf_1, ipf_2, 50, 50);
```

Note: the weights (50, 50) reflect the relative contribution of each function to the coupled function.

The OF_Coupler Tool

Typical usage of the coupler in code:

```cpp
01 IvPFunction *ipf_1 = zaic_1.extractIvPFunction();
02 IvPFunction *ipf_2 = zaic_2.extractIvPFunction();
03
04 OF_Coupler coupler;
05 IvPFunction *ipf = coupler.couple(ipf_1, ipf_2, 90, 10);
```

Note: the weights (50, 50) reflect the relative contribution of each function to the coupled function.
The OF_Coupler Tool

Typical usage of the coupler in code:

```c++
01 IvPFunction *ipf_1 = zaic_1.extractIvPFunction();
02 IvPFunction *ipf_2 = zaic_2.extractIvPFunction();
03 OF_Coupler coupler;
04 IvPFunction *ipf = coupler.couple(ipf_1, ipf_2, 35, 65);
```

Note: the weights (50, 50) reflect the relative contribution of each function to the coupled function.

A Side By-Side Look
Reflector Tools
For Objective Functions with Multiple Dependent Variables

<table>
<thead>
<tr>
<th>Optimization Problems</th>
<th>Multi-Objective Optimization</th>
<th>Robot IvP Helm Optimization</th>
<th>The IvP Domain</th>
<th>IvP Functions</th>
<th>IvPBuild</th>
<th>ZAIC Tools</th>
<th>IvPBuild</th>
<th>Reflector Tools</th>
</tr>
</thead>
</table>

**The Reflector Tool**

The Reflector Tool creates IvP Functions over multiple coupled decision variables.

**Question: What are coupled decision variables?**

In the below 2D objective function, the merits of the heading decision may be evaluated without also simultaneously considering the speed decision. 

… because the function was built by joining the two independent (decoupled) functions:
The Reflector Tool

The Reflector Tool creates IvP Functions over multiple coupled decision variables.

Question: What are coupled decision variables?

In the below collision avoidance objective function, the merits of the heading decision may NOT be evaluated without also simultaneously considering the speed decision.

The below IvP Function is created with the Reflector Tool.

The IvPBuild Toolbox

General Usage Pattern

Recall The IvPBuild Toolbox Pipeline:

1. Create a BuildTool instance.
   - Build Tool
2. Pass IvPDomain to the tool.
   - IvPDomain
3. Pass Parameters to the tool.
   - Params
4. Extract the IvP Function.
   - IvPFunction

Create the ZAIC instance, passing the overall IvPDoman and particular variable.

Set the desired ZAIC parameters.

Extracting the IvPFUnctio from the ZAIC tool.
The IvPBuild Toolbox
Using the Reflector Tool

With the reflector tool, the build pipeline has the additional step of passing a pointer to the underlying function to be approximated.

1. Create a BuildTool instance.
2. Pass IvPDomain to the tool.
4. Pass Parameters to the tool.
5. Extract the IvP Function.

The Reflector Tool
Example Code Usage

A typical code structure (usually found in the implementation of an IvP Behavior)

```c
AOF_Gaussian aof(ivp_domain);
aof.setParam("xcent", 50);
aof.setParam("ycent", -150);
aof.setParam("sigma", 32.4);
aof.setParam("range", 150);

OF_Reflector reflector(aof);

int pieces_created = reflector.create(1000);
IvPFunction *ipf = reflector.extractIvPFunction();
```

Create an underlying objective function given an IvP Domain
Parameterize the underlying function
Create a reflector
Direct the reflector to create an approximation with 1000 pieces
Extract the objective function
The default usage of the Reflector is to specify a desired number of pieces.

```c
int pieces_created = reflector.create(1000);
```

It will return the actual number of pieces generated.

The number of pieces specified by the caller depends on:
- **Accuracy** desired.
- **Time** budget for creating the objective function.

100 pieces

625 pieces

2500 pieces

- The appeal of pure uniform function generation is that it is easy to use.
- No insight needed regarding the underlying function form.

- The drawback is that it is potentially very inefficient.
- Some areas of the function domain may be very well approximated with very few pieces.
The Reflector Tool

Directed Refinement

- The Directed Refinement option of the Reflector Tool allows the caller to refine a given area of the domain with a given uniform pieces size.

**Basic idea:**

- Make an initial uniform function
- Identify a sub-area of the domain.
- Apply a smaller uniform piece to this area

**Typical code structure using the Directed Refinement option.**

```plaintext
01 OF_Reflecter reflector(aof);
02 reflector.setParam("uniform_piece", "discrete @ x:5,y:5");
03 reflector.setParam("refine_region", "native @ x:10:24,y:-25:20");
04 reflector.setParam("refine_piece", "discrete @ x:2,y:2");
05 reflector.create();
06 IvPFunction *ipf = reflector.extractIvPFunction();
```

- Create a reflector passing it the underlying function
- Specify the initial uniform pieces size
- Specify the sub-region to refine
- Specify the piece dimensions used in the refine region
- Invoke the algorithm
- Extract the objective function
The Reflector Tool

Pros and Cons of Directed Refinement

• The appeal of directed refinement is that it is very efficient in its use of pieces and sampling of the underlying function.
• Less pieces, greater accuracy over pure uniform functions.

• The drawback is that it requires the user to actually have insight into the form of the underlying function.
• Sometimes this is the case, often it is not.

• There is another option: Smart Refinement

Smart Refinement

• The Smart Refinement option of the Reflector Tool allows the caller to automatically identify pieces that may need further refinement.

Basic idea:

1. Make an initial uniform function

2. Maintain a fixed-size priority queue of pieces with poor regression scores

3. Continually pop the piece with the worst score and refine (split) the piece into smaller pieces. (up to some # of pieces)
The Reflector Tool

Smart Refinement

Typical code structure using the **Smart Refinement** option.

```
01 OF_Reflector reflector(aof);
02 reflector.setParam("uniform_amount", 1000);
03 reflector.setParam("smart_amount", 400);
04 reflector.setParam("refine_thresh", 0.5);
06 reflector.create();
08 IvPFunction *ipf = reflector.extractIvPFunction();
```

- Create a reflector passing it the underlying function
- Specify the initial amount of uniform pieces
- Specify the additional amount of pieces dedicated to smart refinement
- Optionally specify a regression threshold to abort smart refinement early
- Invoke the algorithm
- Extract the objective function

Pros and Cons of Smart Refinement

- **The appeal of smart refinement** is that it is efficient in its use of pieces and sampling of the underlying function.
- **Less pieces, greater accuracy** over pure uniform functions.
- **Does not require the user to know anything** about the underlying function.

- The drawback is that its regression scores are not always accurate.

- **In short, Smart Refinement** is very powerful and convenient if used in conjunction with other refinement, and not relied on too heavily.
THE END