Optimization
or
Computational Solution Techniques for Mathematical Programming Problems

A Brief Survey of Problems and Exact and Heuristic Solution Methods

Nick Bennett, Bob Robey, Tom Robey, and Jon Brown

Supercomputing Challenge Kickoff, 2011-2012

Why Do We Build Models/Simulations?

(Your answers)

When you have a working model and you are getting ready to present your project remember your answers to this question. Does your project satisfy the reasons that you have given? Are you expecting the judges to figure out what you have discovered from your model or are you explaining to them what you have learned from your model?

Optimization and Models

Optimization is a formal way (among many others) for extracting information from a model. Used in this way it is not the main purpose for the project. What type of information does it produce?

Optimization can also take a major role in the project in studying which optimization approaches work best. In this role the results compare optimization approaches.

Past projects that used optimization:

2009-2010 3rd place team: Los Alamos High School
- To Kill a Flocking Bird (Peter Ahrens, Stephanie Djidjev, Vicky Wang, Mei Liu)
  - Applied brute force, bracketing, steepest descent and genetic optimization algorithms to a NetLogo model of flocking birds.

2010-2011 2nd Place Team: Los Alamos High School
- BrilliAnts (Peter Ahrens, Dustin Tauxe, Stephanie Djidjev)
  - Investigated Ant Colony Optimizations

Basic Definitions

- Optimization, or mathematical programming, is the study and practice of seeking, in a systematic way, the maximum or minimum values of a function (the objective function), and the values of the decision variables (the inputs to a given function) where the maximum or minimum objective function values are found. Complicating this in practice is that decision variables are often subject to constraints, and that the nature of the objective function may make analysis difficult.

- An algorithm is a step-by-step problem solving procedure. Algorithms can range from very simple (e.g. cooking recipes, driving directions between two locations on a map, the Sieve of Eratosthenes for enumerating prime numbers) to moderately complicated (e.g. Runge-Kutta methods of numerical integration) to very complicated (e.g. algorithms to maximize potential profits from oil-drilling activities).

Basic Definitions (cont.)

- The *feasible region* is the set of decision variable values which satisfy all of the constraints in a mathematical programming problem. *Constraints* define the limits of the feasible region.
- A *maximum or minimum* (the plural forms are *maxima* and *minima*), is a point in the feasible region where the value of the objective function can't be increased (for a maximum) or decreased (for a minimum) by moving in any direction in the local neighborhood. If this condition holds for the entire feasible region, then the point is a *global maximum* or *global minimum*.; otherwise, it's a *local maximum* or *local minimum*.
- *Extremum* (pl: *extrema*) can be used to refer to either a minimum or maximum.
- *Optimum* (pl: *optima*) is a minimum or maximum (usually global), as relevant to the stated problem. For example, if we're trying to minimize the objective function, then the optimum is the point at which the objective function is minimized.

Basic Definitions (cont.)

- An *exact algorithm* is one which is mathematically guaranteed (under some stated conditions) to arrive at the solution to a particular type of problem. For some problems, there are known exact algorithms which are computationally very efficient; for others, all known exact algorithms are computationally expensive – prohibitively so, in some cases.

- *NP-complete* – computationally expensive usually requiring an exhaustive search of all possibilities. NP is nondeterministic polynomial time. *NP-hard* is similar complexity but without a formal proof.

- A *heuristic algorithm* is one which incorporates certain rules of thumb, or experience-based rules. It ignores whether the algorithm can be mathematically proven correct. In most cases heuristics require fewer mathematical and logical operations than exact algorithms – though there is no guarantee that the heuristic will arrive at the same solution.

Optimization

- Sometimes more of an art than a mathematical science
- There is no “perfect” optimization method
- Powerful – worth $$$
  - A 1% savings in operation costs can increase profit by 10%
- Used in nearly every business, engineering, science or professional field
- Many advances in the last decade (compare to most math disciplines with little change in 100 years)

This makes it great material for a SC Project

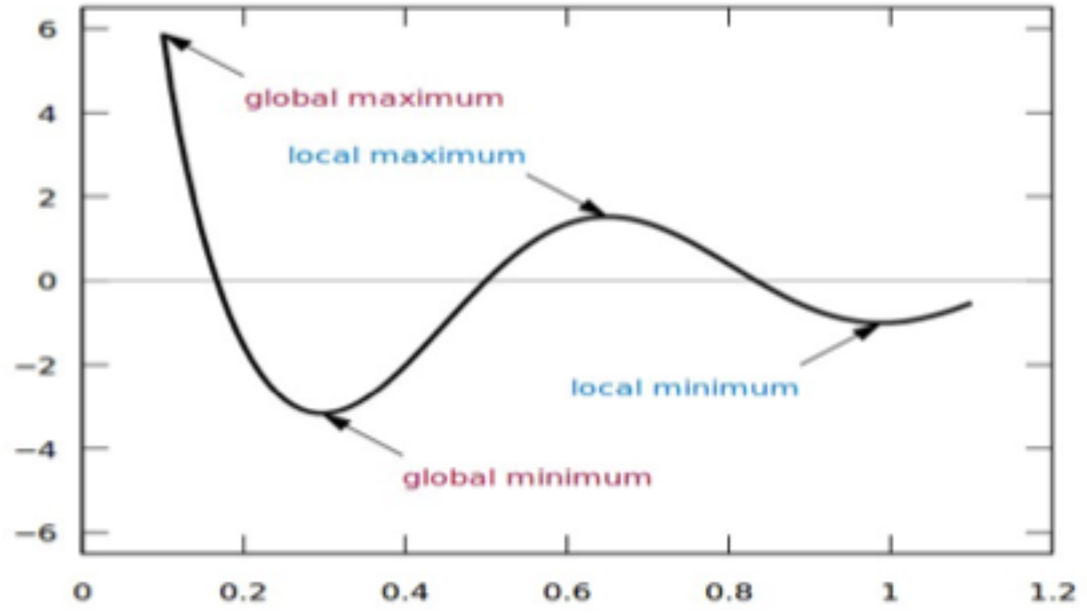
First Step: What is “Good”?

- Determine what you are optimizing - must be quantifiable
    - Less fuel, less time, more widgets
- Often called an objective function, cost function, fitness function, or an evaluation function
- Evaluation functions should be written where the optimum is a maximum or a minimum

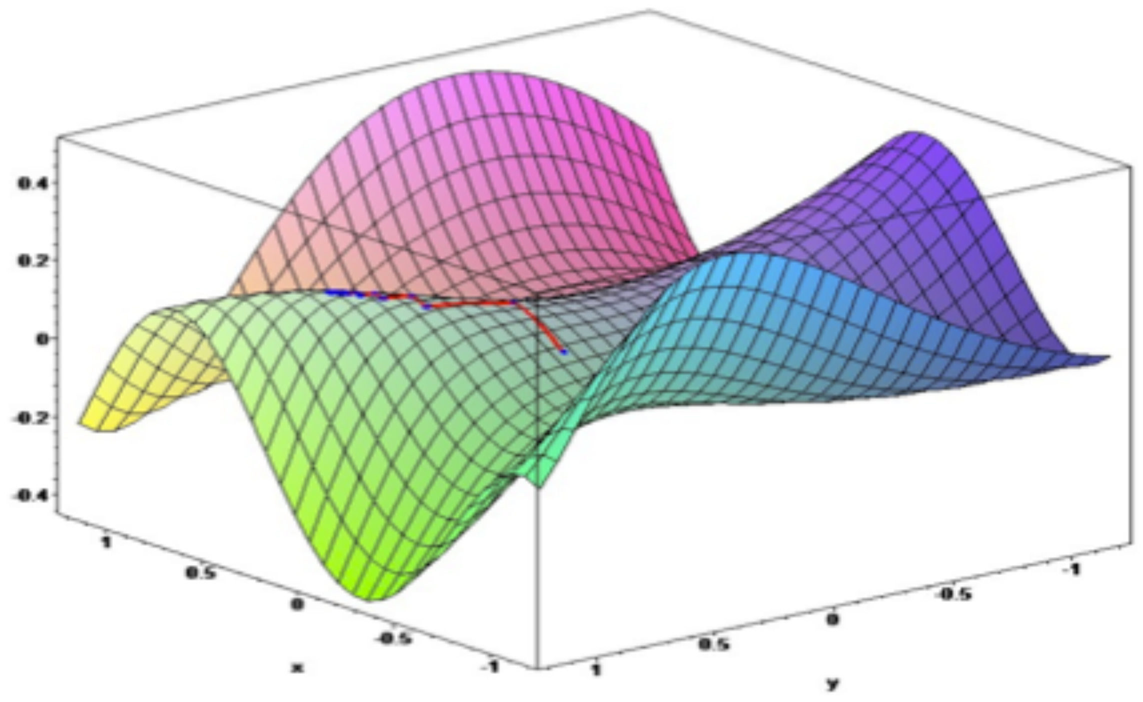
A minimum will be assumed for the rest of this presentation

Second Step: What are the Independent Variables?

- One variable



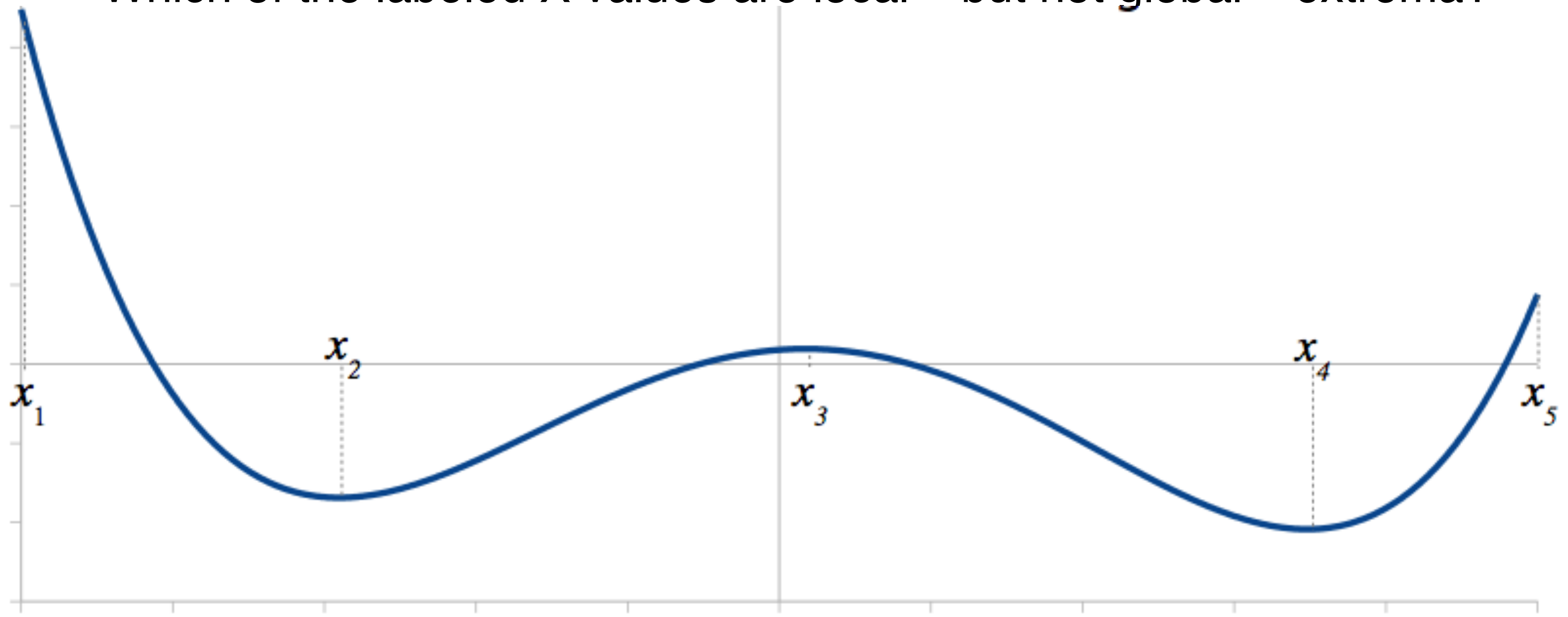
- Two variables



Example: Global vs. Local Extrema

Consider the following graph. Assume that the decision space is represented by the X values (limited to the visible width of the graph), and that the objective function values are the Y values.

- At which of the labeled X values does the graph reach its minimum and maximum Y values? These are the global minimum and global maximum (i.e. global extrema).
- Which of the labeled X values are local – but not global – extrema?



Is the Optimization Problem Well-Posed?

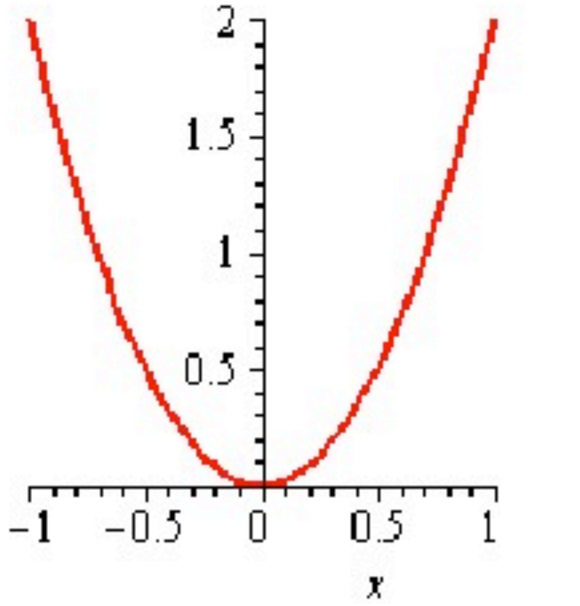
1. A solution exists
2. It is unique
3. The solution depends continuously on the data so that the solution can be found from nearby data.

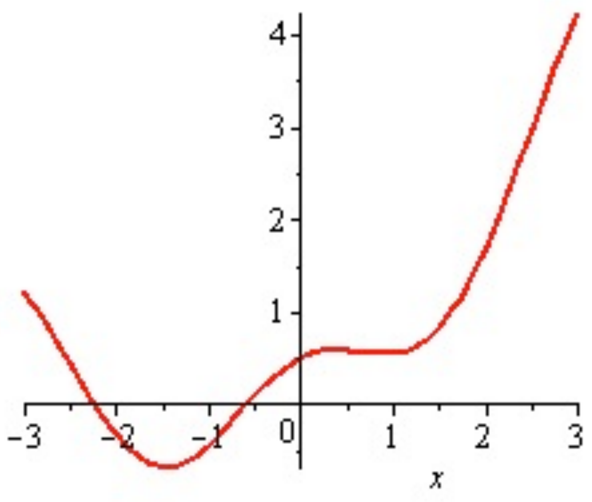
   Example: $f(x) = 1,\ x \neq \prod;\ f(\prod) = 0$

   Can a computer ever really have a variable that equals $\prod$?

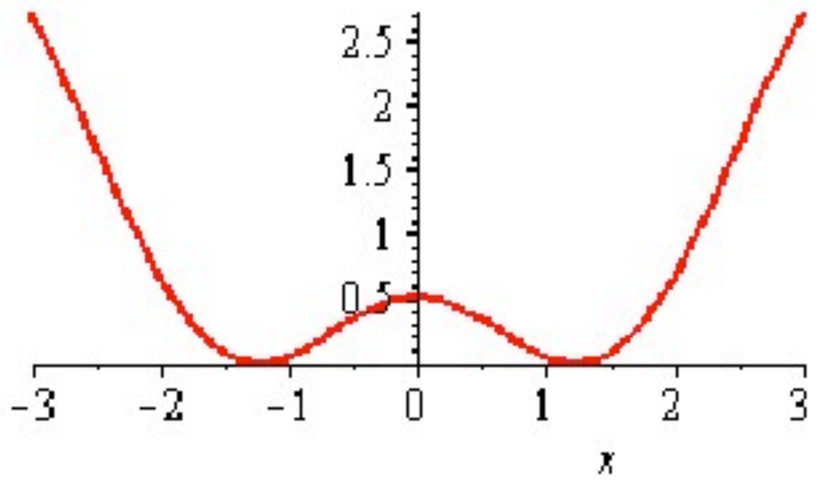
Uniqueness

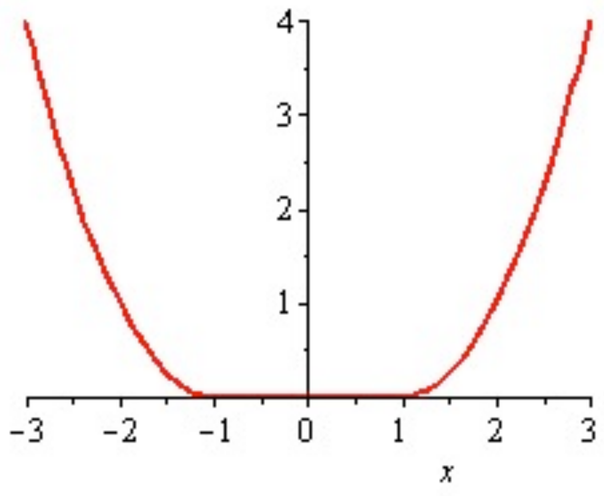
Unique Solution





No Unique Solution





Techniques for dealing with ill-posed problems are discussed in Appendix B

Considerations

- Methods work best with smooth functions
- Independent variables should not themselves be correlated. If they are, as they often are, the optimum values of the independent variables may be wildly off. Yet the minimum of the objective function will be reasonable.
- There is no guarantee in most of the methods of a global minimum, just a local minimum

Tricks

- Isolate independent variables and solve
  - Similar to technique in “Clue”®
- Example – Objective function is highest level of ant activity. Independent variable are hours of daylight and temperature.
- Take data at same temperature but on different length days

What is a Good Method?

- Method should evaluate the function as few times as possible or fewest number of steps/guesses
- Parallel methods can work better with more guesses at each step rather than the least number of evaluations. For parallel methods we should redefine steps to consist of one or more guesses or function evaluations
- A good method for one problem is not necessarily a good method for another problem
- Robustness (i.e. working for all cases) can be more important than efficiency or fewest steps

Types of Problems

- Linear – treated in Operations Research, Business Schools. A function example is 22 lbs of plastic per 100 widgets produced. (Note: Operations Research also delves into most of the following types of problems also.)

- Non-linear – occurs more often in science where inter- relationships between variables are more complex. A small change in an independent variable can generate large changes in the objective function.

- Complex problems (or NP-complete) – a solution can be verified, but there is no known efficient way to locate a solution. Examples are the traveling salesman problem or the knapsack problem. While an optimum solution cannot be easily computed, a near-optimal solution (good enough solution) can be found. Computing a "perfect" chess move is a good example of this type of problem.

Simple Mathematical Optimization

- Optimum must be at a point where the slope is zero. In math this is expressed as when the derivative is zero. This point is a local minimum. However, it doesn't necessarily follow that any such zero-slope point is a global optimum for the function.
- The points where the slope is zero are the set of candidate points. Some adjustments must be made
  - Points that violate the constraints must be discarded (such as negative values that violate physical reality)
  - Add to the set of points the intersections of the optimization surface and the constraints.
- We now examine this set to see which are global extrema, and which of those is a minimum extrema.

Example: Maximize Fenced-in Area

Imagine that you own a piece of land, located at the bottom of a cliff. You intend to fence in a portion of this land, as a pen for animals.

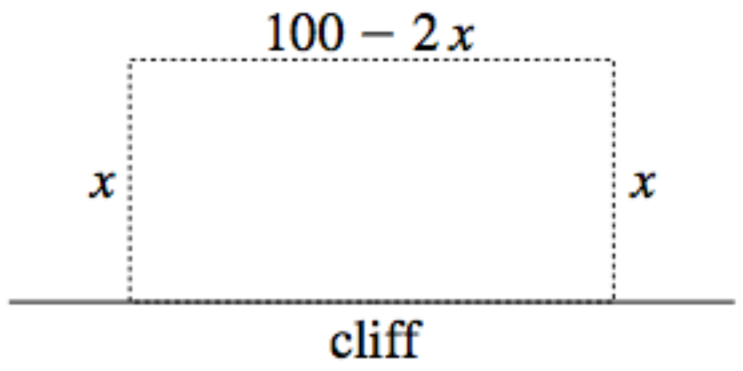
- You have enough material for 100' of fence.
- For simplicity, you've decided to make the pen rectangular in shape.
- For economy, you're going to use the cliff wall as one side of the rectangle; you don't need to put fencing material on that side.

Your task is to find the dimensions for the fenced-in area that maximize the area enclosed, using only the materials you have.

- What mathematical function describes the area enclosed by the fence, in terms of its dimensions?
- Are there any constraints on the dimensions? What are they?

Example: Maximize Fenced-in Area (cont.)

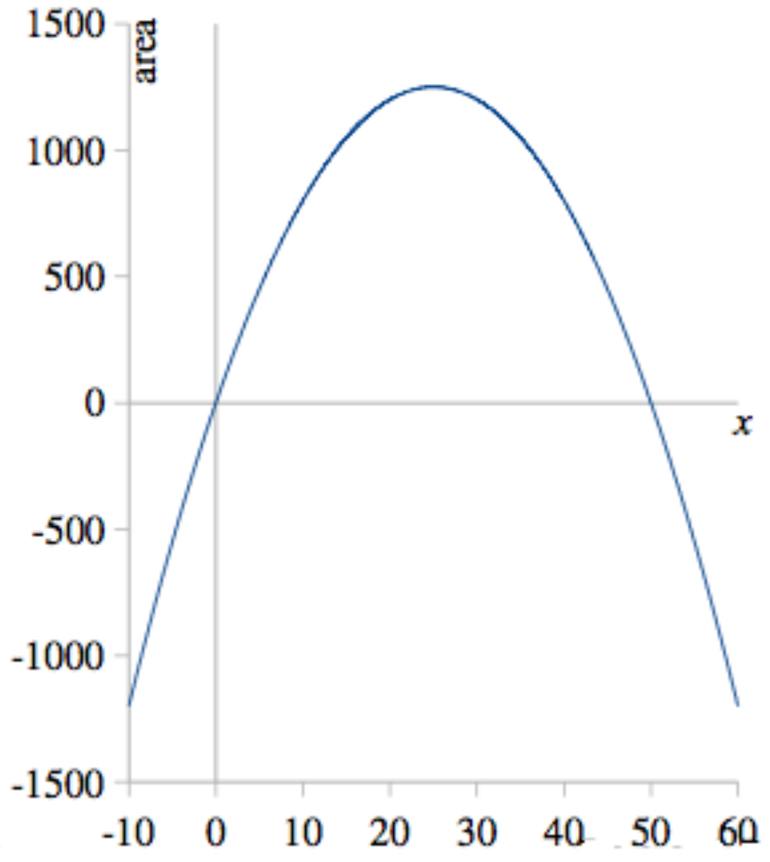
We can draw a simple diagram of the problem, as follows:



$$\text{area} = f(x) = x(100 - 2x) = 100x - 2x^2$$

Our task is to find the value of x that gives the largest value for $f(x)$. We recognize $f(x)$ as a quadratic of the form $f(x) = ax^2 + bx + c$. From intermediate algebra, we remember that the vertex of the parabola described by the quadratic is found where $x = -b/2a$.

In this case, $b = -100$, and $a = 2$. Computing x and the resulting area, we get the maximum area (1250 ft^2), when the dimensions are 25' X 50'.



Three Main Approaches

- ***Direct solution*** – used for linear, quadratic and other simpler problems where simple analytic approaches are workable.
    - Simplex Method or Linear Programming are typical solution techniques. Also Lemke's algorithm for quadratic programming, the stepping-stone algorithm for the transportation problem, the Hungarian method for the assignment problem, and Dijkstra's algorithm for finding the shortest path on a network
- ***Iterative solution methods*** – take slight perturbations of the initial guess and search in the direction with the better value.
    - Steepest Descent and Conjugate Gradient solvers are examples of methods to solve these problems
- ***Directed random search methods*** – search algorithms have randomness as a key element, where the sampling distributions are influenced by the objective function, and where the amount of randomness tends to decrease as the number of iterations increase.
    - Simulated Annealing, Genetic Algorithms (GA), Ant Colony optimization, and Particle Swarm Optimization are some of the methods most suitable for this class of problems.

Direct Methods
Linear Programming via the Simplex Method

When the objective function is linear, all the constraints are linear, and the decision variables are continuous, we have a linear programming problem (LP). In 1947, George Dantzig invented the simplex method to solve linear programming problems. In very general terms, this algorithm proceeds as follows:

- Set up the problem as a system of linear equations, with more variables than equations (new variables are added as needed, to turn inequalities into equations).
- By a series of matrix "pivoting" operations, move from one feasible solution to another, where each successive solution improves the objective function.
- When the objective function cannot be improved further, we've found the optimum.

This process is as a series of moves from vertex to vertex, along the edges of an n-dimensional convex polytope, with each move improving the objective function. When none of the edges lead in a direction that improves the objective function, we've found the optimal solution.

Exercise: Maximize Profit on Exports[1]

A firm exports two types of machines: P and Q. Type P occupies 2m^3 of space, and type Q requires 4m^3. The mass of type P is 9kg; type Q masses 6kg. The total available shipping space is 1,600m^3 and the total mass of the machines cannot exceed 3,600kg. The profit on type P is \$100 and the profit on type Q is \$80. How many of each machine must be exported to maximize profit, and what is that maximum profit?

Here's the LP formulation, along with a graphical view of the feasible region (i.e. the region containing all points satisfying the constraints):

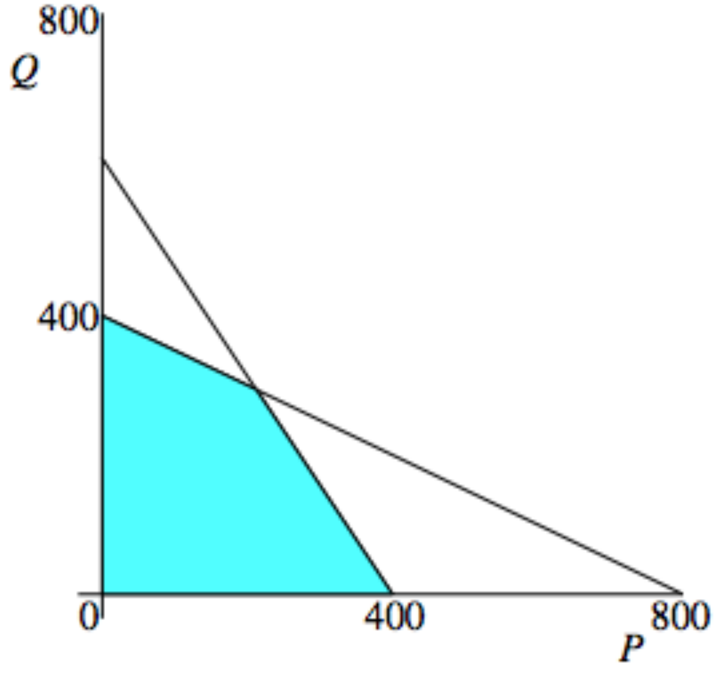
$p =$ number of type P machines exported
$q =$ number of type Q machines exported

Maximize:

$$100p + 80q$$

Subject to:

$$\begin{array}{rcrcl} 2p & + & 4q & \leq & 1600 \\ 9p & + & 6q & \leq & 3600 \\ p & & & \geq & 0 \\ & & q & \geq & 0 \end{array}$$



Exercise: Maximize Profit on Exports (cont.)

We could easily solve this example by computing the profit at all four vertices of the feasible region and selecting the vertex with the highest profit. The advantage of the simplex method is that it almost never needs to visit each vertex; it always moves along the edge that improves the objective function most. This is a big advantage when problems are much larger than this one.

There are many libraries that implement the simplex method. For this example, we'll use Apache Commons Math (http://commons.apache.org/math), an open source Java library.

1. In NetBeans, open the SimplexExample project.
2. Open the `SimplexExample.java` file from the org.nm.challenge.optimization source package.
3. Complete the code by following the instructions that begin on line 72 of `SimplexExample.java` (hint: review the constants declared in lines 42-48).
4. Compile and run the program. What's the result?
5. Is the answer integral? If not, how should we interpret it?

Exercise: Giapetto's Woodcarving

Giapetto's Woodcarving Inc. manufactures two types of wooden toys: soldiers and trains. A soldier sells for \$27 and uses \$10 worth of raw materials. Each soldier that is manufactured increases Giapetto's variable labor and overhead costs by \$14. A train sells for \$21 and uses \$9 worth of raw materials. Each train built increases Giapetto's variable labor and overhead costs by \$10. The manufacture of wooden soldiers and trains requires two types of skilled labor: carpentry and finishing. A soldier requires 2 hours of finishing labor and 1 hour of carpentry labor. A train requires 1 hour of finishing and 1 hour of carpentry labor. Each week, Giapetto can obtain all the needed raw material but only 100 finishing hours and 80 carpentry hours. Demand for trains is unlimited, but at most 40 soldier are bought each week. Giapetto wants to maximize weekly profits (revenues – costs).

Operations Research: Applications and Algorithms, 4th Edition, by Wayne L. Winston (Thomson, 2004).

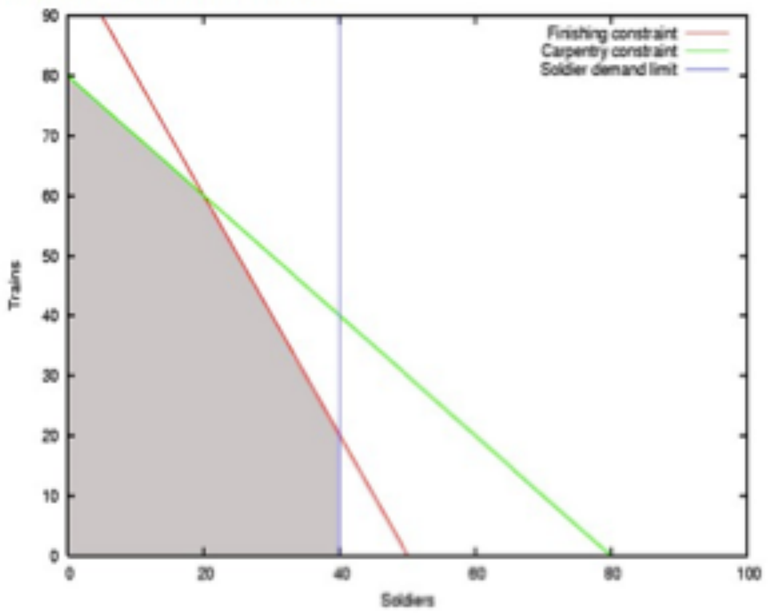
More detail on this problem at:
http://www.ibm.com/developerworks/linux/library/l-glpk1/

Exercise: Giapetto's Woodcarving

To summarize the important information and assumptions about this problem:

- There are two types of wooden toys: soldiers and trains.
- A soldier sells for \$27, uses \$10 worth of raw materials, and increases variable labor and overhead costs by \$14.
- A train sells for \$21, uses \$9 worth of raw materials, and increases variable labor and overhead costs by \$10.
- A soldier requires 2 hours of finishing labor and 1 hour of carpentry labor.
- A train requires 1 hour of finishing labor and 1 hour of carpentry labor.
- At most, 100 finishing hours and 80 carpentry hours are available weekly.
- The weekly demand for trains is unlimited, while, at most, 40 soldiers will be sold.

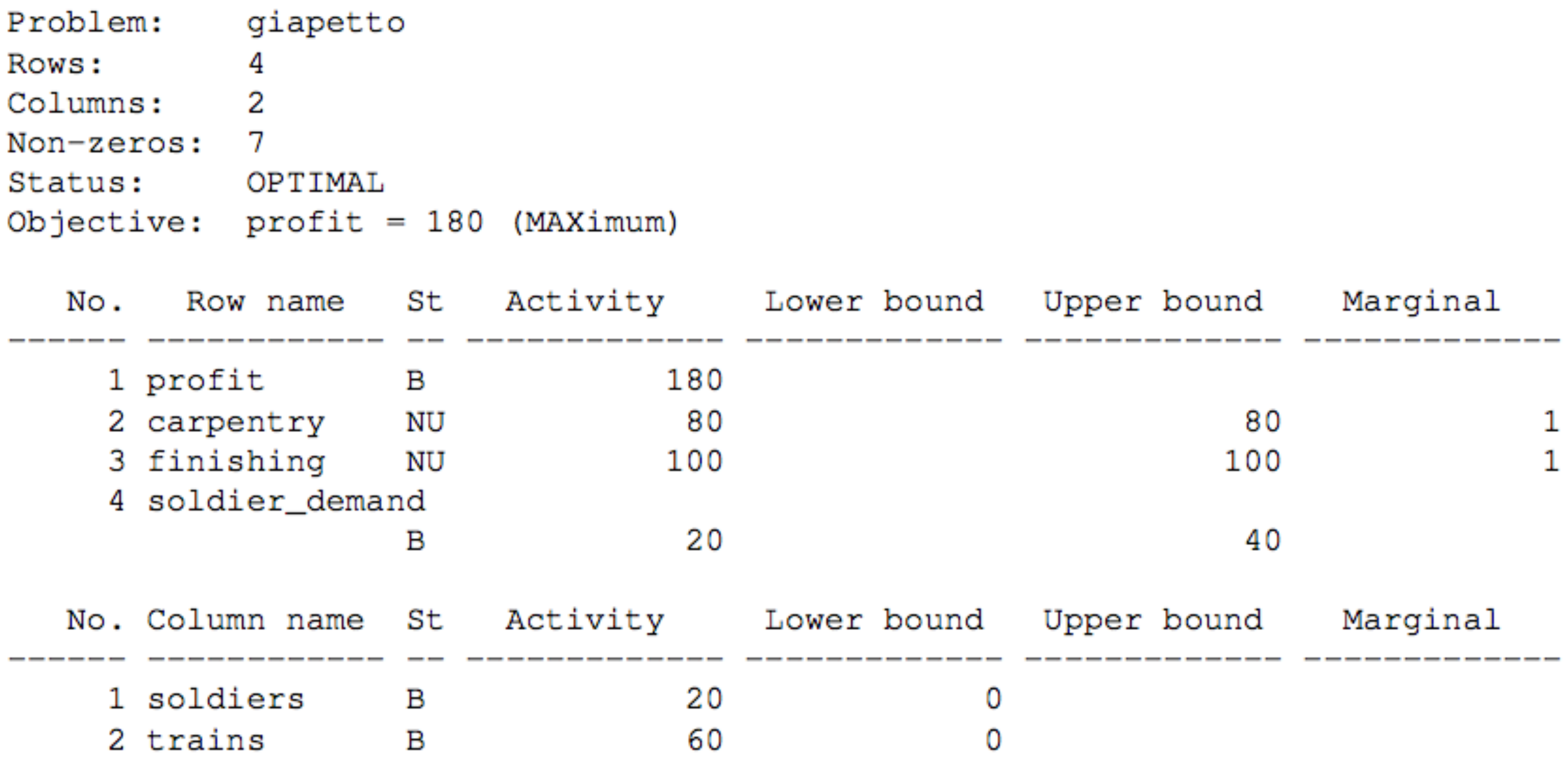
Figure 4. Giapetto's feasible region



The goal is to find the numbers of soldiers and trains that will maximize the weekly profit.

Exercise: Giapetto's Woodcarving

1. Install libgplk0, libglpk-dev, glpk-utils, glpk
2. Copy the file giapetto.mod from the giapettoLP directory
3. glpsol -m giapetto.mod -o giapetto.sol



```
Problem:    giapetto
Rows:       4
Columns:    2
Non-zeros:  7
Status:     OPTIMAL
Objective:  profit = 180 (MAXimum)

   No.   Row name   St   Activity     Lower bound   Upper bound    Marginal
------ ------------ -- ------------- ------------- ------------- -------------
     1 profit       B            180
     2 carpentry    NU            80                          80             1
     3 finishing    NU           100                         100             1
     4 soldier_demand
                    B             20                          40

   No. Column name  St   Activity     Lower bound   Upper bound    Marginal
------ ------------ -- ------------- ------------- ------------- -------------
     1 soldiers     B             20             0
     2 trains       B             60             0
```

Specialized Linear Programming Variations

Some optimization problems are better suited to more specialized methods than the general-purpose simplex method. Examples include:

- The transportation problem: Given *n* sources, each with a supply s_n of some commodity; *m* destinations, each with a demand d_m for that commodity; a cost c_{ij} for transporting one unit of the commodity from source *i* to destination *j*; find the transportation scheme that satisfies the demand at minimum cost. Solution techniques for this problem exploit the network nature of the problem to solve it more efficiently.

- The assignment problem: Given *n* resources, and *m* tasks (where $m \geq n$), where there is a cost c_{ij} associated with assigning resource *i* to task *j*, and where each resource must be assigned to exactly one task, and each task may be assigned a maximum of one resource, find the set of assignments that minimizes the total cost. The most well-known technique for solving this type of problem is the Hungarian method.

- In general, problems where some or all decision variables must have integral values often require specialized algorithms. This is especially the case where only the values 0 and 1 are allowed.

Interior Point Solution Techniques

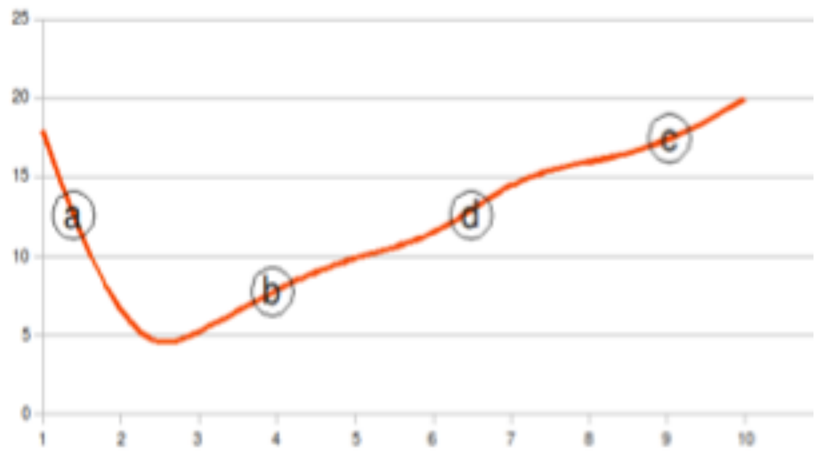
While the simplex method and its variants are usually very good, and they're widely used, there are extreme instances where the simplex method gives close to worst-case performance. The first practical alternative to the simplex method for these problems was Karmarkar's algorithm, invented in 1984.

Karmarkar's algorithm can be viewed as moving through the interior of the polytope, in a relatively small number of steps. For this reason, we refer to this and similar methods as *interior point methods*.

The computational cost of formulating LPs for interior point methods, and of computing each step through the interior of the polytope, are very high, but the number of iterations required for solution is low. This trade-off makes these techniques well suited to certain types of very large problems, but poorly suited to most others.

One-dimensional Methods: Bracket methods

- For single variable cases with unimodal objective function (no local minima)
- Take 3 starting points widely separated. The middle point, b, must be lower than the other two. Then take a point in between a-b or b-c. Suppose we take a point in b-c called d. If d > b, then the new bracket triplet is a < b < d, else, the new triplet is b < d < c.
- The various methods use different ways to estimate the best point to take for the next guess. Golden mean method uses the golden ratio. Other methods use a parabolic function or first derivatives.



Exercise 1 – Bracket Method

Find the minimum of f(x) = cos(x) + 1, which occurs at x = ∏. The starting interval is (0,6), with an initial guess for the minimum of 2.

1. Install libgsl0ldbl, libgsl0-dev
2. Open up Eclipse. Select New Project, New C Project, Executable:Empty Project. Put in name of project – Bracket. Click through rest of windows.
3. Right click on Bracket in Project Explorer. Select new: C Source file. Name it bracket.c. Enter source from BracketExample/bracket.c or http://www.gnu.org/software/gsl/manual/html_node/Minimization-Examples.html
4. Right click on Bracket again and select Properties all the way down at the bottom. Open up C/C++ Build and select settings. Select GCC C Linker:Libraries. Add the libraries gsl and then gslcblas. Compile project.
5. Select Run menu: Run configurations. Click on add (small page with plus sign at upper left). Setup run configuration and run problem.
6. See the Appendix for platform specific instructions.

Try the other minimization algorithms in the library, golden section and quad golden. The reference page is http://www.gnu.org/software/gsl/manual/html_node/Minimization-Algorithms.html. The function names are gsl_min_fminimizer_goldensection and gsl_min_fminimizer_quad_golden. Which works best for this problem?

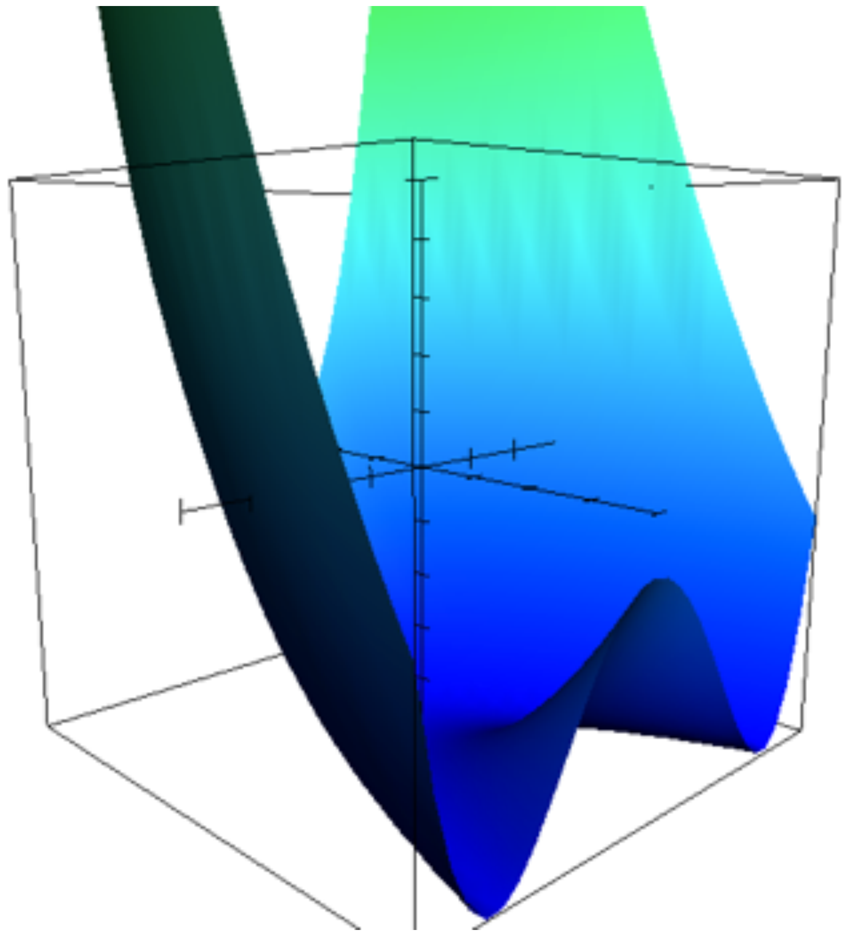
Exercise 2 – Bracket Method

Take your favorite Netlogo model. Let's say that it is a fire egress model. For one slider, optimize the number of people that escape. Note that at each input value, you will have to take a statistical average of the result because of the random number used in the simulation.

Multivariable Methods
Rosenbrock Test Case

If you have a Mac, open Grapher (Applications -> Utilities -> Grapher). Create a 3-D plot. Enter the equation:

$z = (1-x)^2 + 100(y-x^2)^2$



Go to Format -> Axes & Frame. Set the following:

- Abscissa: [-2, 2]
- Ordinate: [-2, 2]
- Height: [0, 2500]

Right click on the plot and edit the appearance. Change Checkerboard to Height. If the plot appears jagged as you play with it then you might want to increase the resolution and reload.

The minimum of this function is at (1,1).

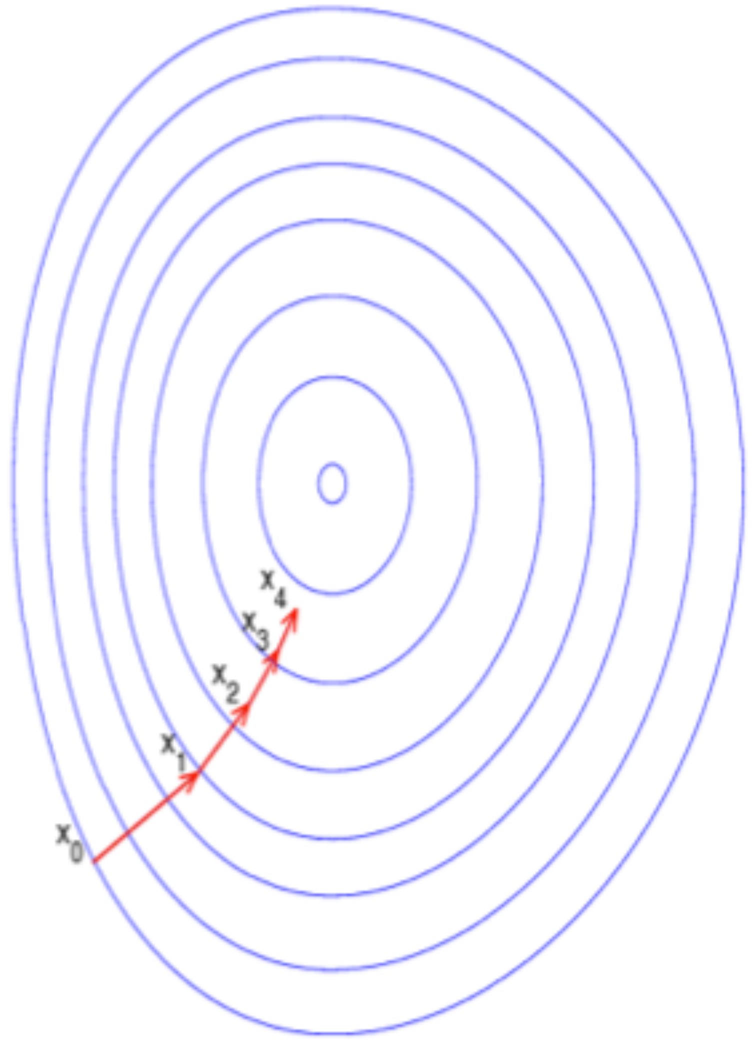
Multivariable Methods: Steepest Descent

1. Multivariable case starts by choosing a direction. A common choice is the steepest descent.

2. Use a one-dimensional line search to decide how far to go in this direction. Do we need to find an optimum for the one-dimensional line search?

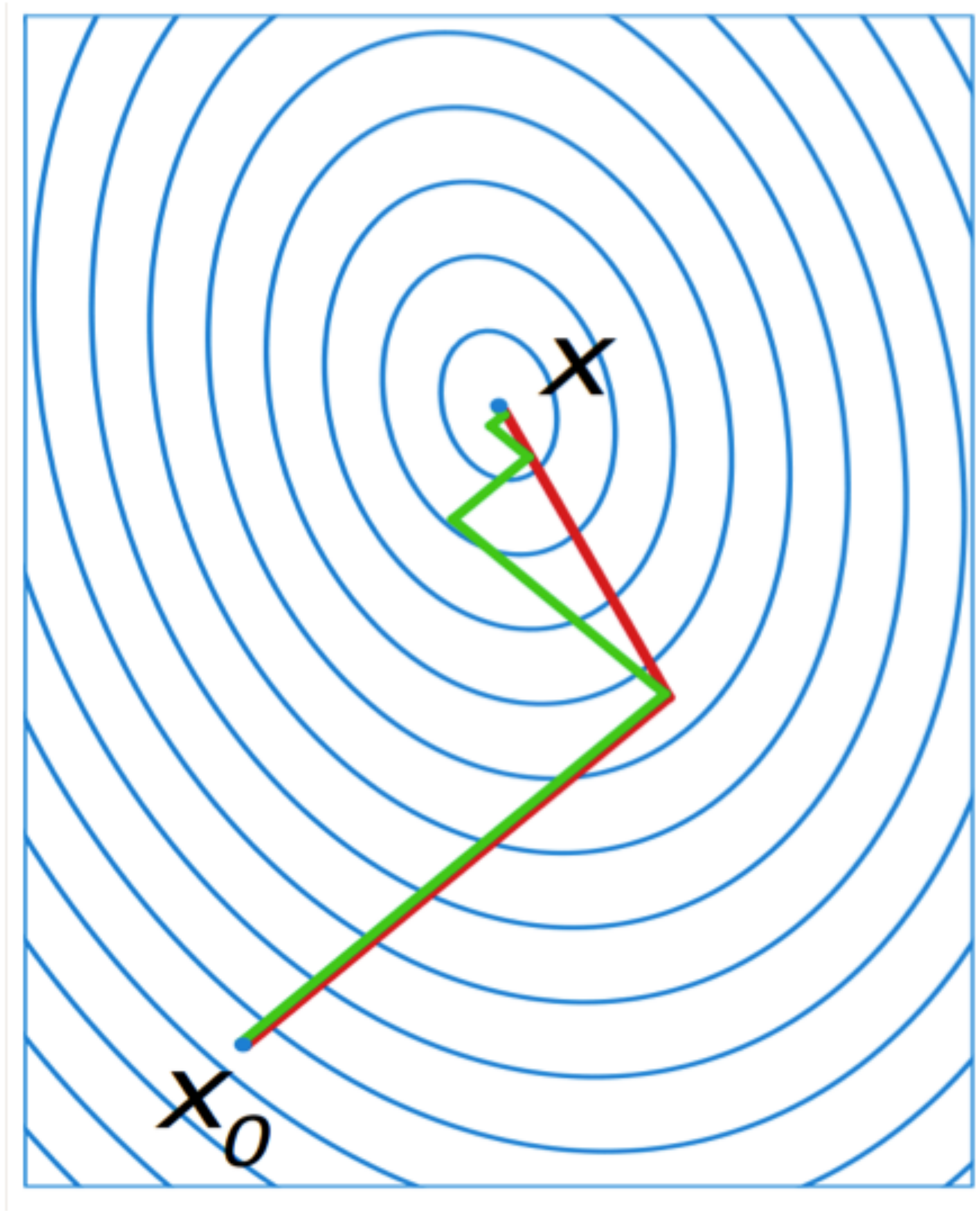
3. Select a new steepest descent direction and repeat the line search. Iterate.

See Numerical Recipes for description, method and sample code.



Multivariable Methods
Conjugate Gradient

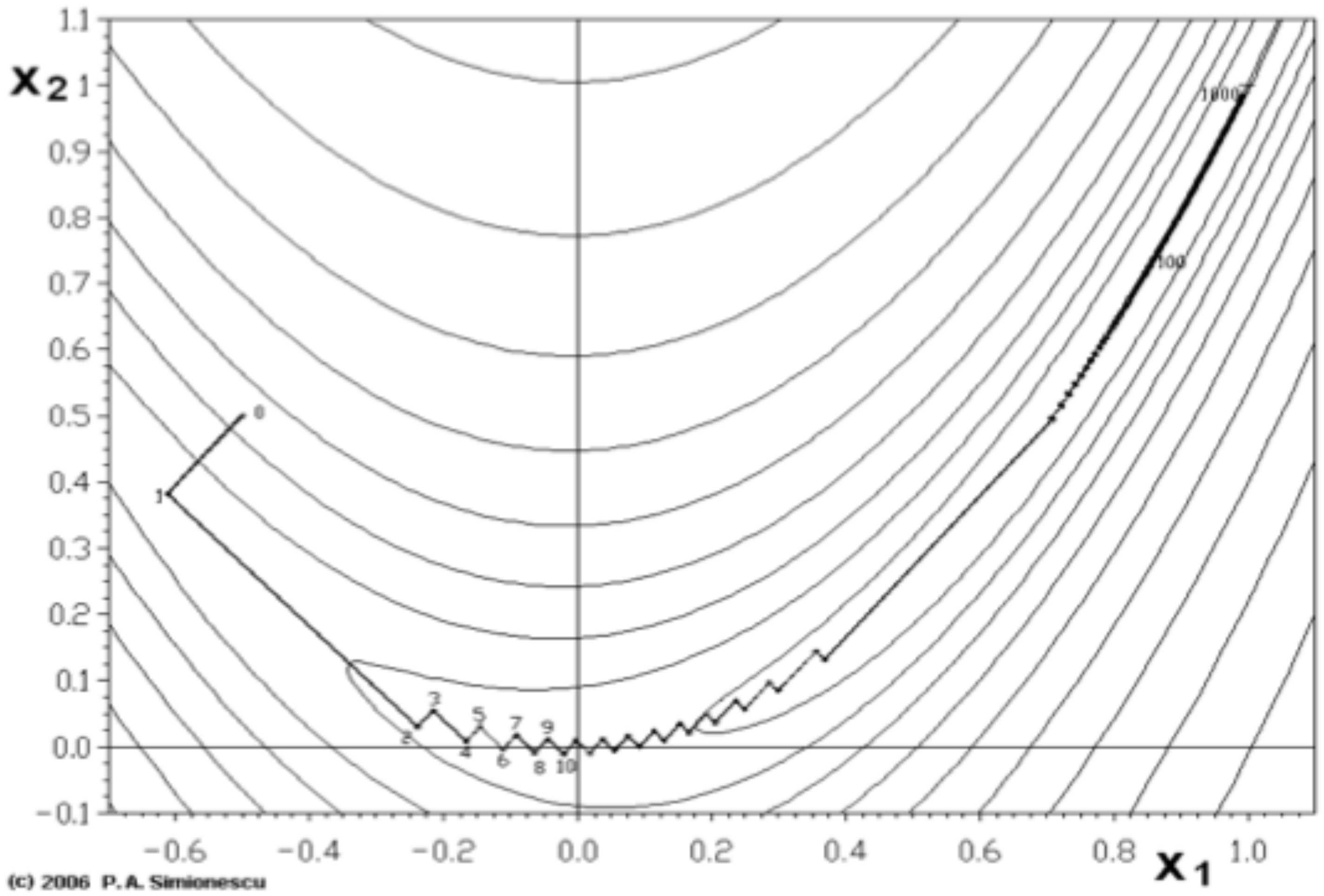
- Trick here is to take the conjugate to all previous search directions to try and force solution "down the valley"
- Why are valleys important? Usually one independent variable is stronger than the other, producing a long valley



Multivariable Methods
Rosenbrock Test Case

Shown here is the steepest descent algorithm applied to the Rosenbrock problem. The narrow curved valley causes the method to take a lot of steps. The conjugate gradient approach was developed to work better on this type of problem. (From wikipedia)

ORNL has pictures of various optimization methods applied to the Rosenbrock and Beale functions: http://www.phy.ornl.gov/csep/mo/node17.html



Example: Conjugate Gradient

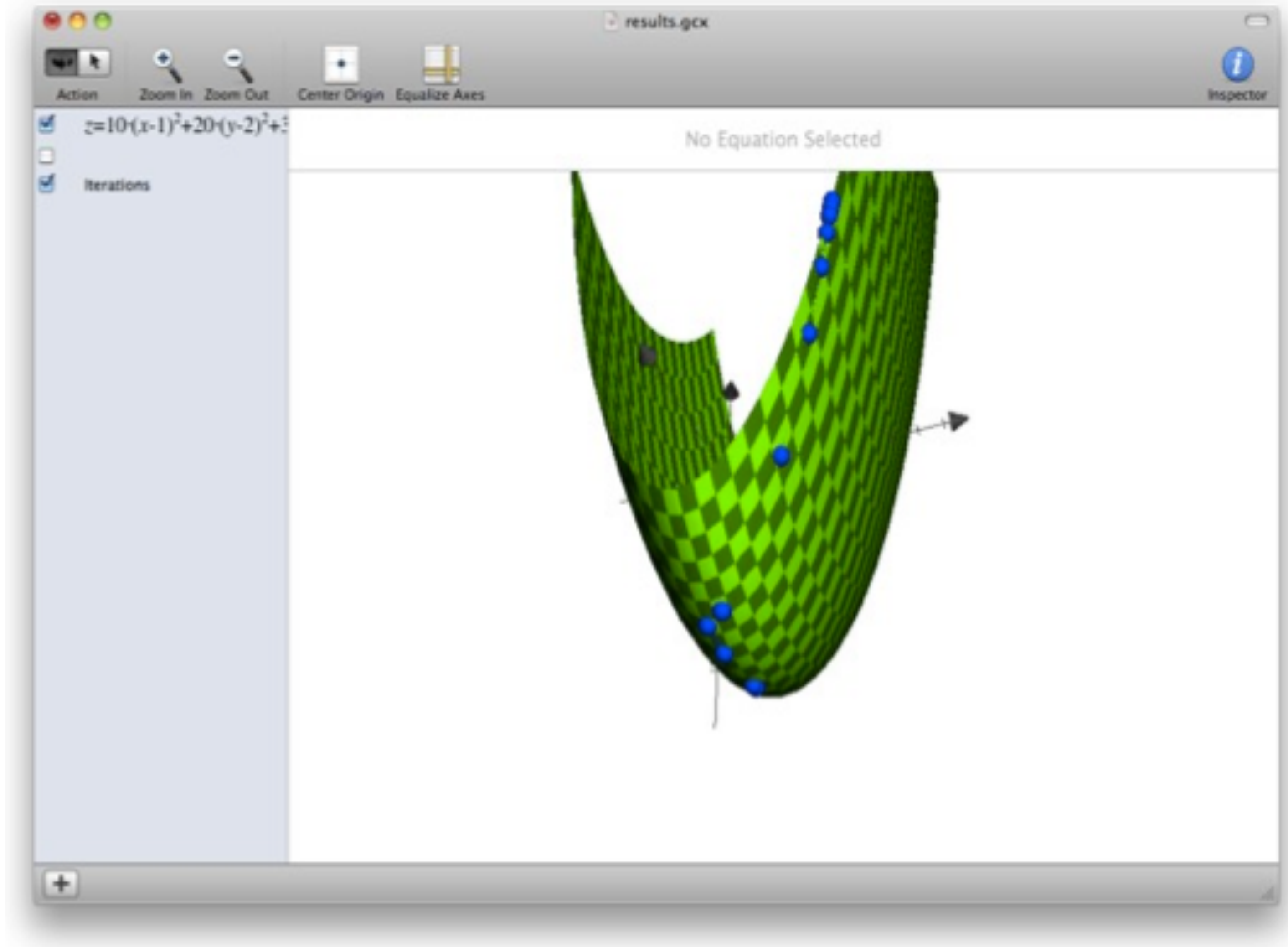
- Install libgsl0ldbl, libgsl0-dev
- Open up Eclipse. Select New Project, New C Project, Executable:Empty Project. Put in name of project – SurfaceMin. Click through rest of windows.
- Right click on SurfaceMin in Project Explorer. Select Import:FileSystem:Kickoff/SurfaceMin. Select both c source files (*.c) and import them.
- Right click on SurfaceMin again and select Properties all the way down at the bottom. Open up C/C++ Build and select settings. Select GCC C Linker:Libraries. Add the libraries gsl and then gslcblas. Compile project.
- Select Run menu: Run configurations. Click on add (small page with plus sign a upper left). Setup run configuration and run problem.
- See Appendix for platform specific instructions.

Graph the results (Mac Only)

- Graph the function by going to Applications->Utilities->Grapher
- select 3D Graph and White and enter the equation:

  $z = 10(x-1)^2+(y-2)^2+30$

- Then go to Format->Axes and Frame and select Height Axis and make it go from 0 to 700.



- Create a text file and cut and paste the results. Using vi, use i to insert and then paste and escape. Then separate the numbers with "," and rows with ";" and then join the rows using "J". Then go to grapher and click on "+" and New Point Set and Edit Points, then import your *.txt file.

Multivariable Methods: A NetLogo Approach

1. Define an objective function *f*.
2. Place all the sliders in the middle of their ranges
3. Either fix random number generators at the middle or use averages of 10 runs.
4. Figure out which inputs are the most important by moving one slider and computing $(f(x_2) - f(x_1))/(x_2 - x_1)$ where x represents the current slider input while the other inputs are held constant. Repeat for all inputs. The larger absolute numbers are the most important inputs at this point. Choose a direction (e.g. move the most important slider two units, the next one unit, etc.).
5. Do a line search in the chosen direction.
6. Go to step 4 and compute new differences and repeat the process until you find a rough optimum input.
7. What inputs are the most important for your model? What does the rough optimum input tell you about your model?

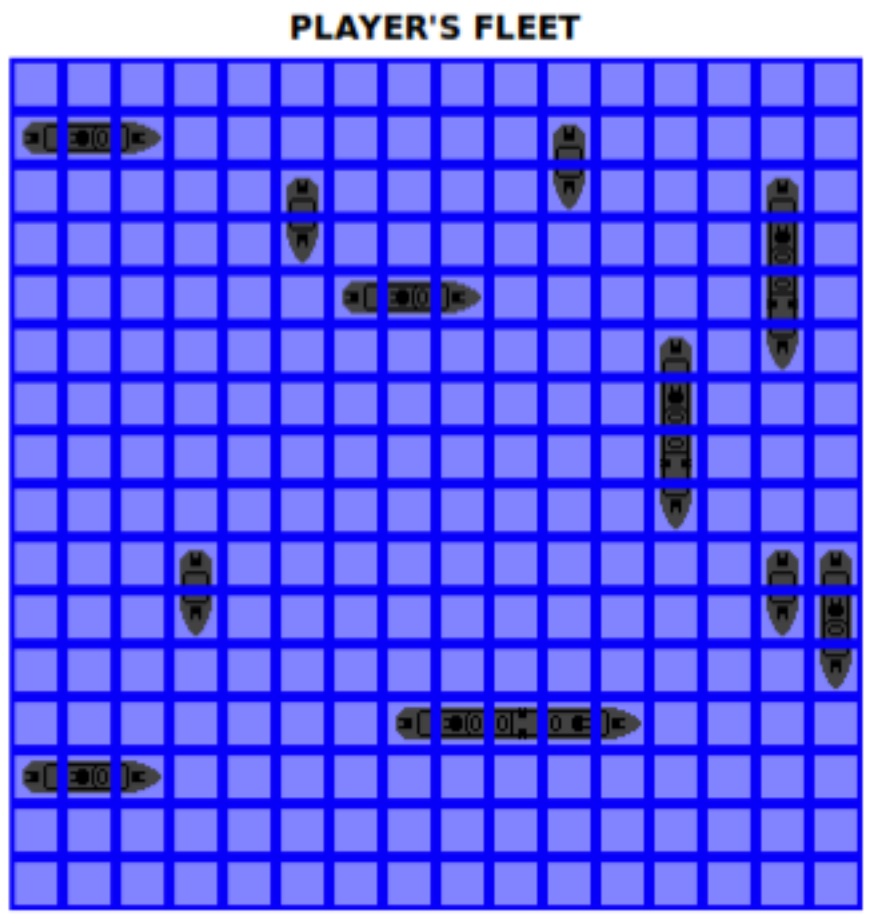
Multivariable Methods: A NetLogo Approach (cont.)

So simplifying this to an extreme:

- Choose the slider that makes the biggest difference.
- Adjust it until you have an optimum.
- Choose the slider that makes the next biggest difference (direction is orthogonal to the first).
- Adjust it until you have an optimum.
- Repeat until you have gone through all the sliders, then start again from the slider that makes the biggest difference.

Directed Random Search Methods

- Best where optimization surface is complex or has local minima
- Has some randomness in the solution technique
- Think of method you use for “Battleship”®
- Heuristic based methods work better than exact algorithms
- Examples of Solution Methods
  - Simulated Annealing
  - Genetic Algorithms
  - Ant Colony Optimization
  - Particle Swarm Optimization



The Traveling Salesman Problem

One example of this is the *traveling salesman problem* (TSP):

Given *n* cities (points), with a distance d_{ij} between cities *i* and *j*,
find the route through all the cities that minimizes the total distance, and returns to the starting point. Exact solution techniques are very expensive once the number of cities moves into the thousands (not that uncommon, when we start seeing such parallels as those between points on a circuit board and cities on a map) and may be years of CPU time.

One of the best heuristics for the TSP is the Lin-Kernighan algorithm, introduced in 1973. This algorithm works by adaptively swapping pairs of tour fragments to formulate an improved tour at each step.

Simulated Annealing

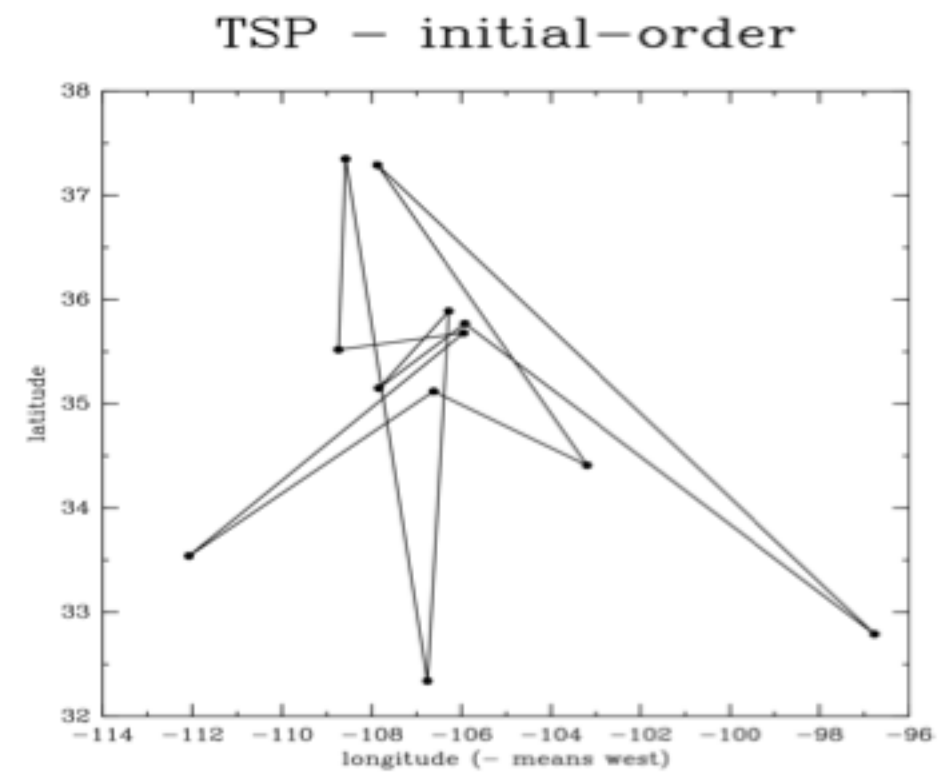
In simulated annealing (which refers to the analogy with how metals cool and anneal), the solution method is allowed to sometimes take an upward step rather than always going in the downward direction. This can be helpful in avoiding local minima and stepping “over the hump” to the global minimum. The frequency of the upward step is always an heuristic algorithm which must be developed for the problem at hand.

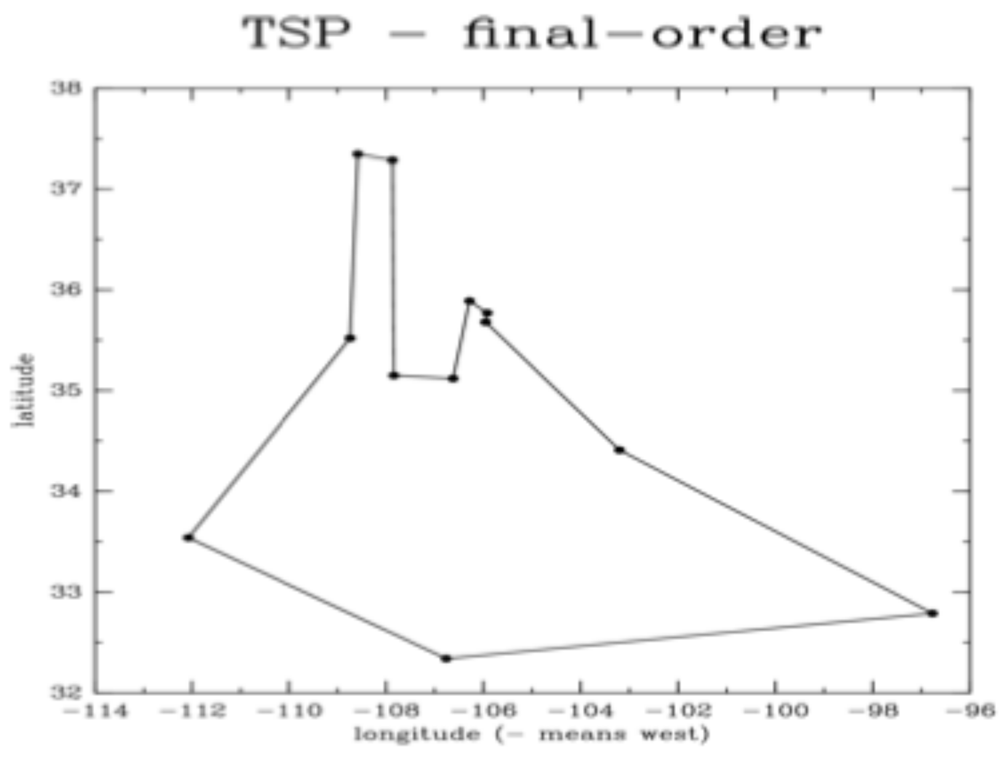
Exercise - Simulated Annealing

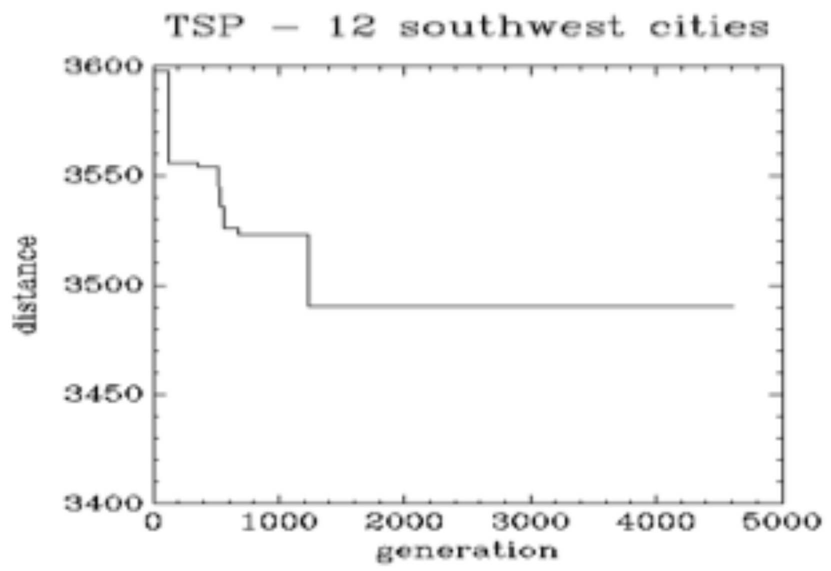
Traveling Salesman Problem

- Open up Eclipse. Select New Project, New C Project, Executable:Empty Project. Put in name of project – Traveling_Salesman_Problem. Click through rest of windows.
- Right click on Traveling_Salesman_Problem in Project Explorer. Select new: C Source file. Name it siman_tsp.c. Enter source from TravelingSalesmanProblem/siman_tsp.c or search the web for siman/siman_tsp.c.
- Right click on Traveling_Salesman_Problem again and select Properties all the way down at the bottom. Open up C/C++ Build and select settings. Select GCC C Linker:Libraries. Add the libraries gsl and then gslcblas. Compile project.
- Select Run menu: Run configurations. Click on add (small page with plus sign at upper left). Setup run configuration and run problem.
- To plot results, run “sh plot.sh” or “sh plotgif.sh”. Look at the files *.eps or *.gif.

Traveling Salesman Problem Results







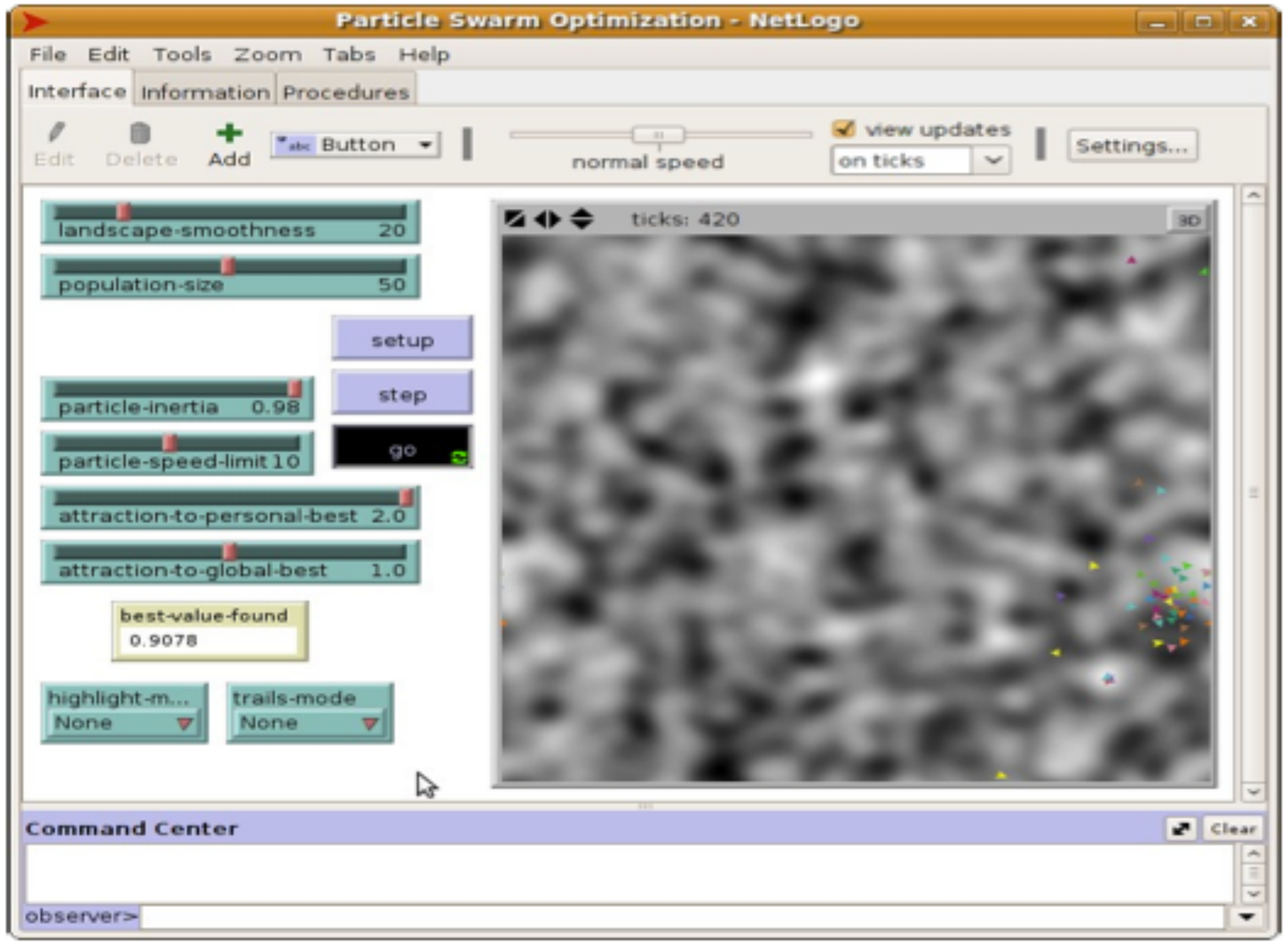
Computational Solution Techniques in Mathematical Programming

Particle Swarm Optimization

Description: http://ccl.northwestern.edu/netlogo/models/ParticleSwarmOptimization

- For particle swarm optimization, each particle is given a position and velocity. Then an acceleration is applied to each particle towards its “personal best” and towards the “global best”. The further away from these locations, the stronger the acceleration towards them. A random factor is also applied to the acceleration forces.

- Exercise – Start up Netlogo 4.1.3. Open up the models library and go to Sample Models:Computer Science:Particle Swarm Optimization (Uri Wilensky, Northwestern University). Setup and run.



landscape-smoothness 20
population-size 50
particle-inertia 0.98
particle-speed-limit 10
attraction-to-personal-best 2.0
attraction-to-global-best 1.0
best-value-found 0.9078
highlight-m... None
trails-mode None

Genetic Algorithms

A genetic algorithm (GA) is a heuristic that emulates some of the mechanisms of evolution to find a good enough solution to a problem. A GA starts with a population of randomly-generated solutions to a problem; each of these is encoded as the genome of an individual member of the population. Then, we follow these steps to produce successive generations of the population:

- Each member of the population is evaluated to assess its fitness
- A subset of the population is chosen to survive to the next generation. This selection is random, but individuals with better fitness are more likely to be selected.
- To replace individuals that don't survive, a subset of the population is chosen (randomly again, but again weighted to the most fit) to reproduce.
- The next generation is produced by combining the genomes from pairs of individuals selected for reproduction.
- Optionally, an offspring's genomes may be mutated slightly in a random fashion.

Exercise: TSP with a Genetic Algorithm

Start up netlogo, v4.1RC5. Open the models library and select Sample Models:Computer Science:Simple Genetic Algorithm(Uri Wilensky, Northwestern University). Set it up and run it.

Convex Hull Construction Heuristic for TSP

One relatively intuitive and effective approach to the TSP begins by finding the subset of points, and the tour connecting them, such that all of the points are either in the subset, or in the interior of the polygon formed by the tour. This polygon is called the *convex hull* of the full set of points. (The concept of the convex hull can be applied to n-dimensional polytopes, but the application to the TSP is primarily of use in two dimensions only.)

Once we have this initial subset and tour (collectively called a *sub-tour*), we add to it progressively, incorporating an additional point at each step. One reasonable heuristic is to select at each step the point that results in the smallest immediate increase in the length of the sub-tour; this type of heuristic is referred to as “greedy”, since decisions are based only on the cost/benefit of immediately available alternatives.

When all points in the set are incorporated into the sub-tour, the algorithm is done.

Exercise: TSP w/ Convex Hull & Greedy Insertion

Go to the site

http://www-e.uni-magdeburg.de/mertens/TSP/node2.html

Try running the second applet. Select a large number of nodes (25) to get a better idea of how the algorithm performs. Then select run. Select solve to see how it compares to the best solution.

Appendices

Appendix A

Linux setup (64 bit Ubuntu)

- Netlogo
  - Run the Netlogo installer
- Eclipse for C programs
  - Install Eclipse by running the eclipse-cpp-galileo-SR1-linux-gtk-x86_64 installer
  - Install Java by running the jdk-6u16-nb-6_7_1-linux-ml.sh installer
  - Install compilers and supporting tools with the package manager
    - build-essential
  - Install libraries and gnu software with the package manager
    - libgsl0ldbl and libgsl0-dev
    - glpk-utils
    - plotutils
  - When setting up the compiling in Eclipse
    - Add to GCC C Linker:Libraries
      - gsl
      - gslcblas
- Netbeans for Java programs
  - Run the netbeans installer

Windows Setup

- Netlogo
  - Run the Netlogo installer
- Eclipse for C programs
  - Install Wascana by running the Wascana installer (http://code.google.com/a/eclipselabs.org/p/wascana/ - download wascana-1.0-setup.exe)
  - Install libraries and gnu software by running gnu installers (http://ascend4.org/Binary_installer_for_GSL-1.13_on_MinGW) for
    - gsl-1.13
  - Add to system path by opening up control panel, search for system environment and adding to the end of the path
    - ;C:\Program Files (x86)\GnuWin32\bin;C:\Program Files (x86)\Wascana\mingw\bin
  - When setting up the compiling in Eclipse
    - Add to GCC C Compiler:Directories
      - "C:\Program Files (x86)\GnuWin32\include"
    - Add to GCC C Compiler:Symbols
      - GSL_DLL
    - Add to MinGW C Linker:Libraries
      - Libraries
        - gsl
        - gslcblas
      - Library search path
        - "C:\Program Files (x86)\GnuWin32\lib"
- Netbeans for Java programs
  - Run the netbeans installer

OSX (Mac)

- Netlogo
  - Place NetLogo in the Applications directory. Double click on the NetLogo icon.
- NetBeans
  - Run the NetBeans installer
- C/C++ Code
  - Install Xcode development tools. These should be on your OS disk or register and download at http://developer.apple.com/technology/xcode.html
  - Install MacPorts (choose the version for your OS): http://www.macports.org/install.php If necessary update using *sudo port -d selfupdate*
  - Make the following directory and cd to it: /opt/local/bin/portslocation/dports/gsl
  - Then install the Gnu Scientific Library (*sudo port install gsl*)
  - Running from the command line
    - Copy SurfaceMin.c and myfunction.c to a directory. Then type the following: *gcc -o multimin SurfaceMin.c myfunction.c -Wall -I/opt/local/include -L/opt/local/lib -lgsl -lgslcblas*
    - To run the program ./multimin

OSX (Mac) (cont)

To run the program in the Xcode IDE:

1. Open XCode (/Developer/Applications/ and drag Xcode.app to your Dashboard.)
2. Then open File/New Project...
3. In the "New Project" Assistant, expand the "Command Line Utility" group.
4. Select "Standard Tool"
5. Click "Next"
6. Give a project name (MultiMin) and directory, then click "Finish".
7. Press Cmd-Shift-R to open the Console window. Output will appear there.

Project->Add to Project and add SurfaceMin.c and myfunction.c. Delete main.c.

Edit Project->Edit Project Settings
Under Search Paths:

Add to User Header Search Paths	/opt/local/include
Add to Library Search Paths	/opt/local/lib/

Under Linking:

Add to Other Linker Flags:	-lgsl -lgslcblas

Set the Architecture to Native Architecture
Click the "Build and Go" toolbar button.

Appendix B:
Ill-Posed Problems

Regularization

Solving ill-posed optimization problems is a process called regularization.

To do this we must introduce additional information into our objective function.

The specific method varies from problem to problem:

- Most physically realistic
- Smallest/largest
- Smoothest

Typical General Strategy

We introduce a new function g(x) which penalizes the features that we wish to remove and replace the old problem:

$$\textit{minimize } f(x)$$

with

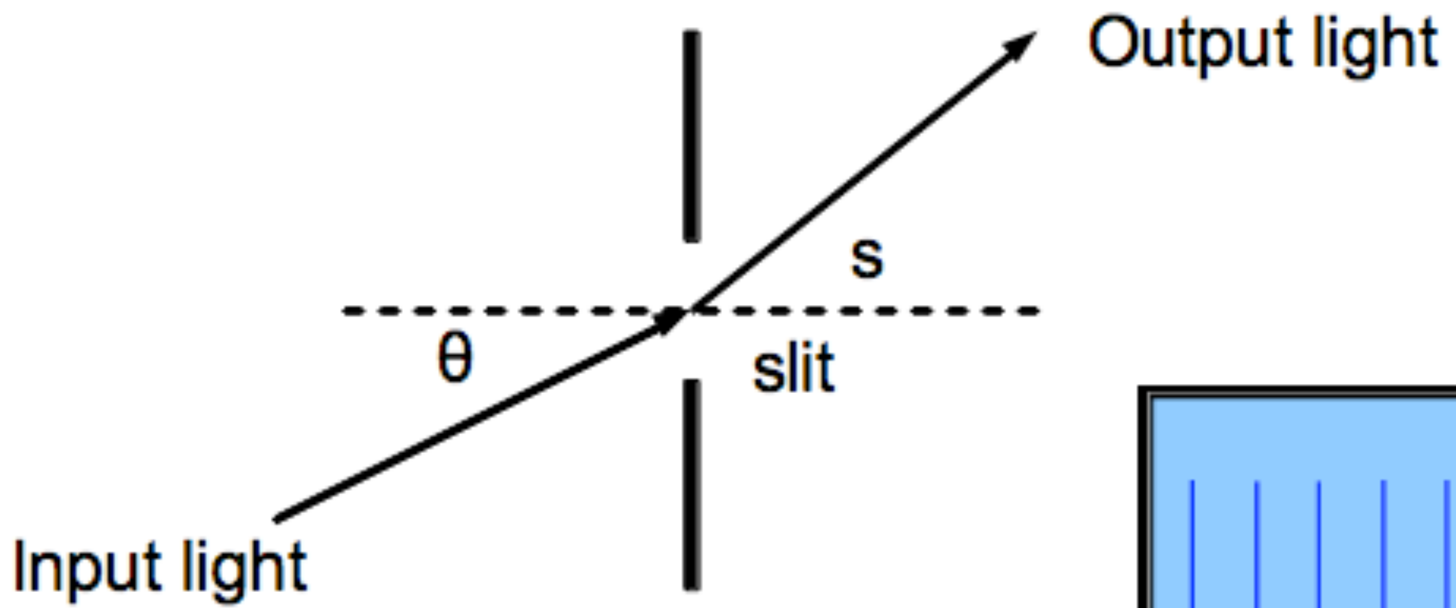
$$\textit{minimize } f(x) + \alpha^2 g(x)$$

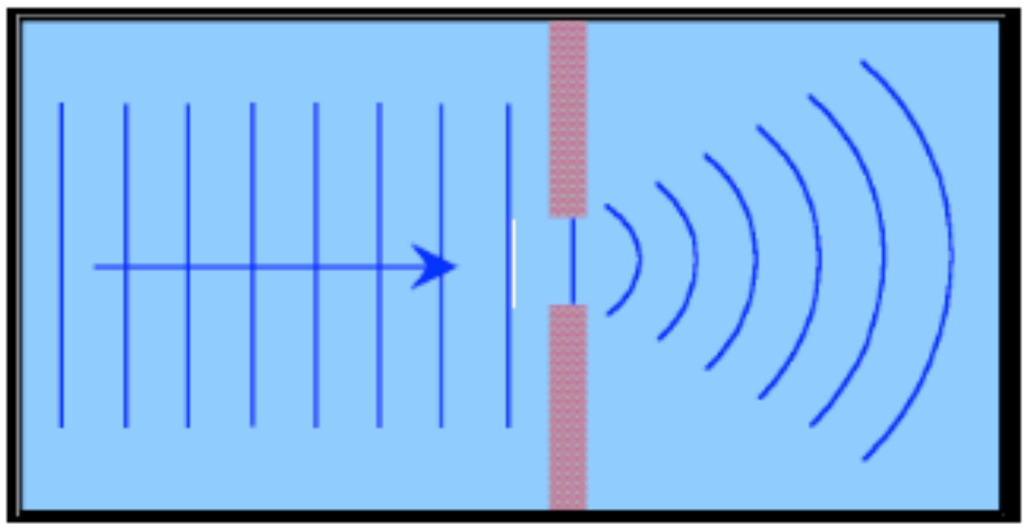
where α is a parameter that determines how much we weight the regularization criteria against solving the problem.
Since we are changing the function that we are optimizing, the solution we get might not be a solution to the first problem, so we tune α to get a “good enough” solution: one that both fits our regularization criteria and (nearly) solve the initial problem.

Why do we need regularization? Example: Shaw Problem

Single slit diffraction problem: light comes in at various angles θ to a slit and diffracts (spreading the light out).

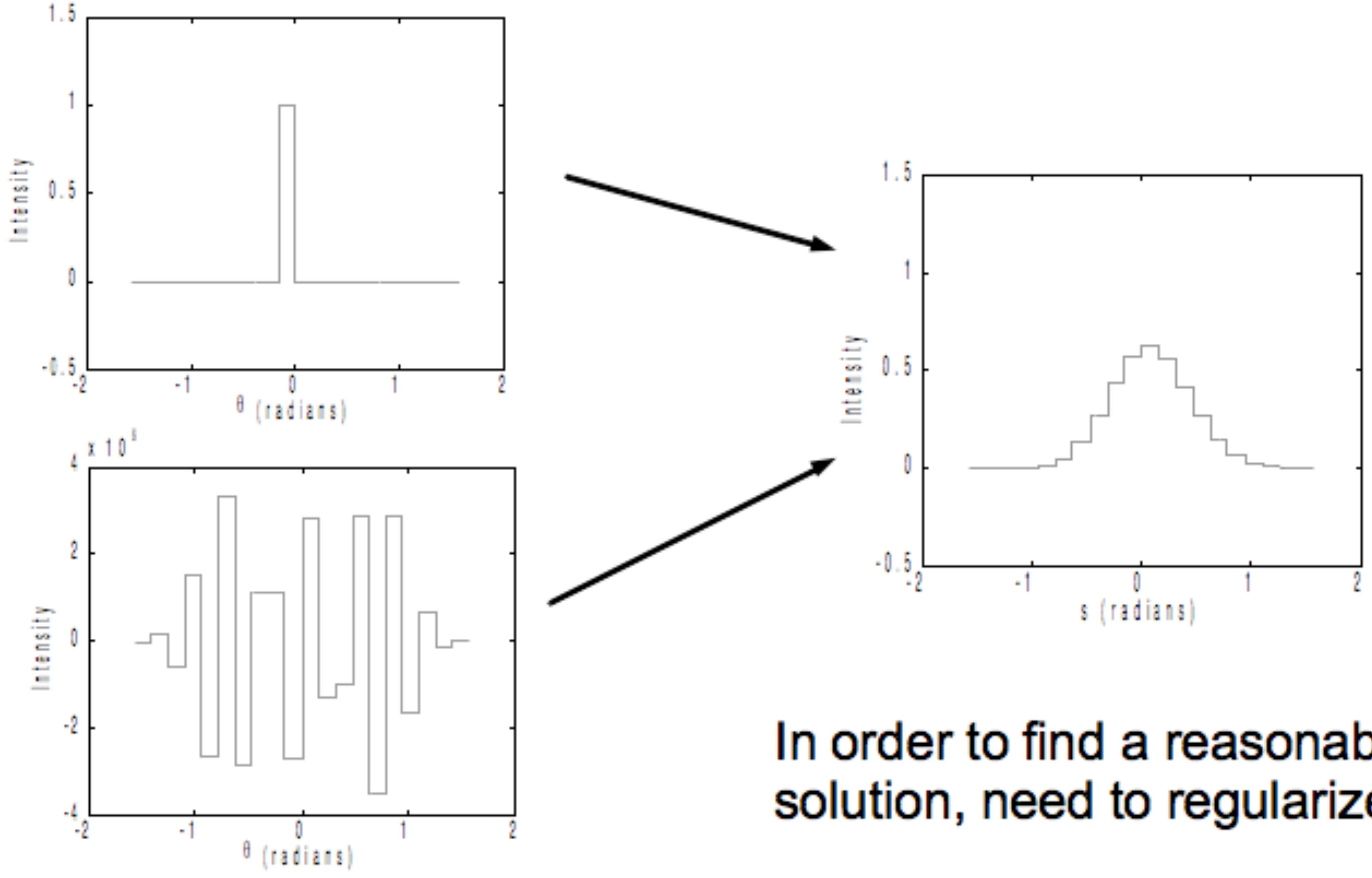
Problem: what input creates a known output?





Shaw Problem Continued

Very different inputs create the same output (to accuracy of 10-6).

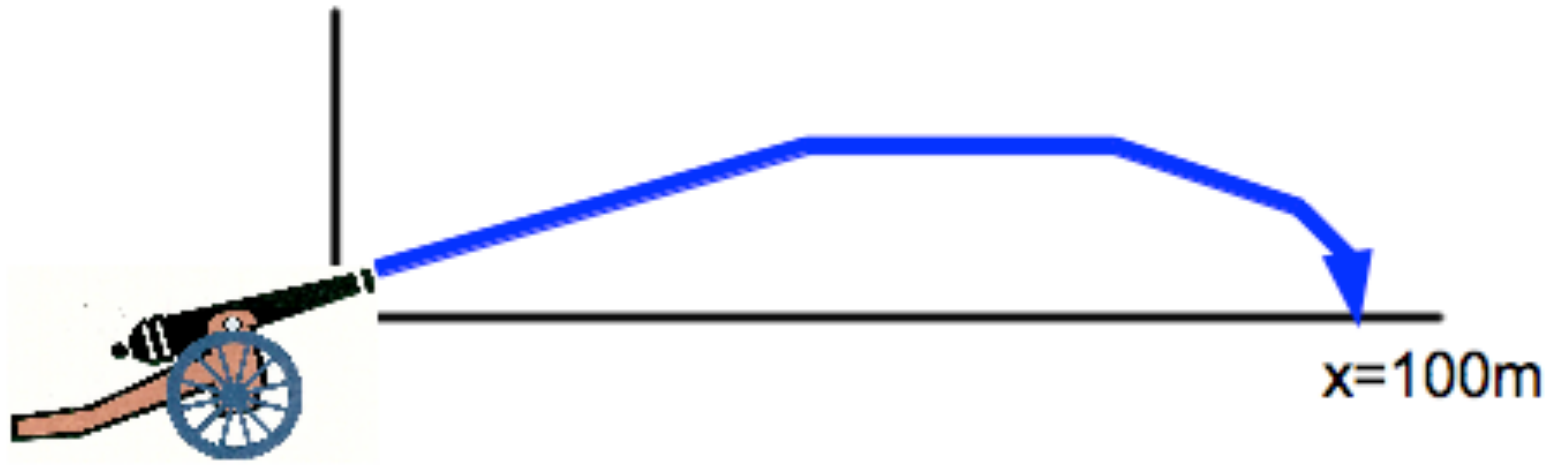


In order to find a reasonable solution, need to regularize!

Worked Example: Projectile Hitting a Target (with air resistance)

Model: input initial velocity v0 and inclination angle θ, and numerically solve Newton's 2nd law to track the position of a projectile (known mass & drag coefficient).
(see projectile.m)

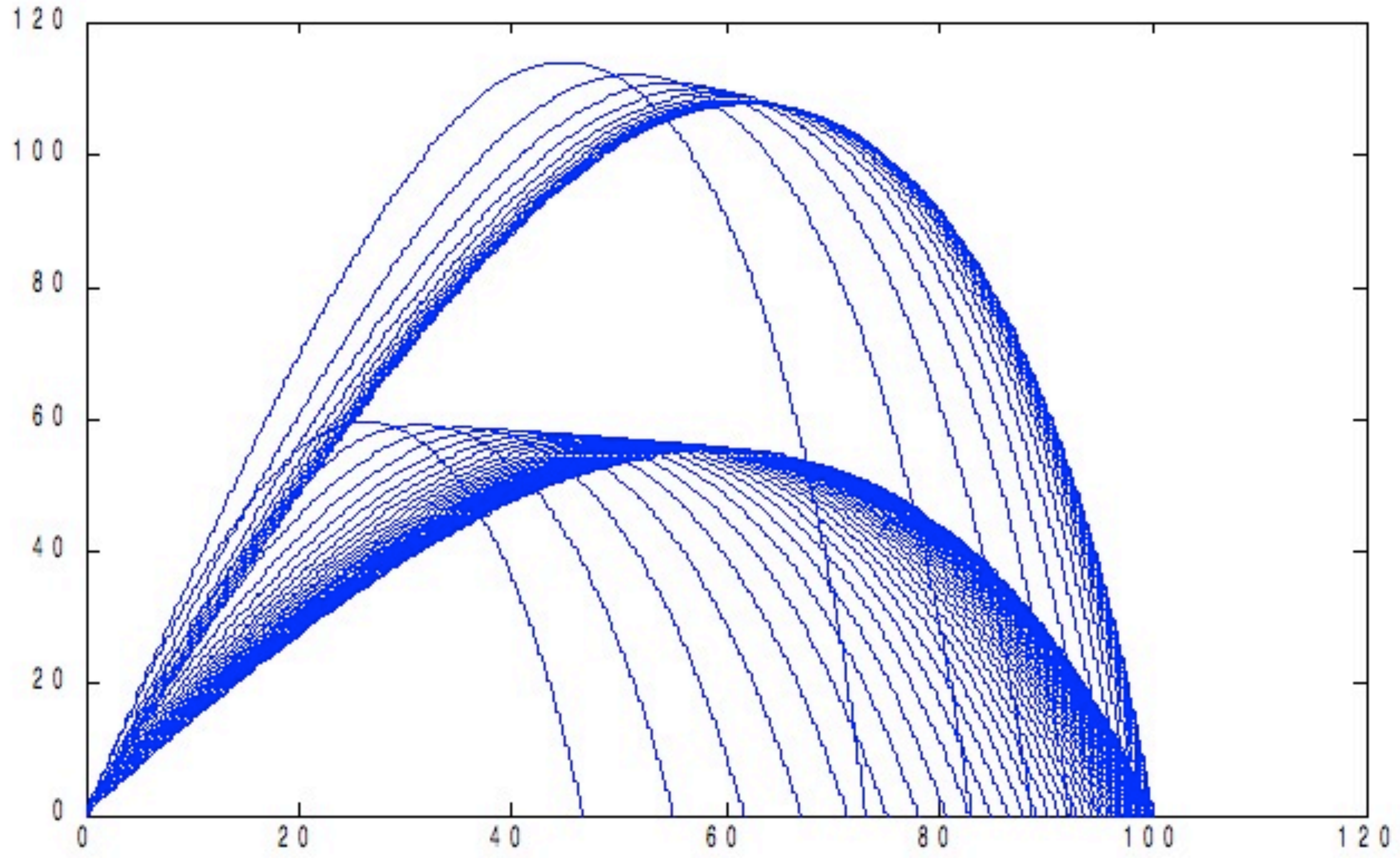
Problem: For what v0 and θ can we hit a target 100 meters away?



Projectile continued

Solution method: steepest descent (steepest.m)

Below: projectile motion at each step of steepest descent algorithm for two different initial conditions.

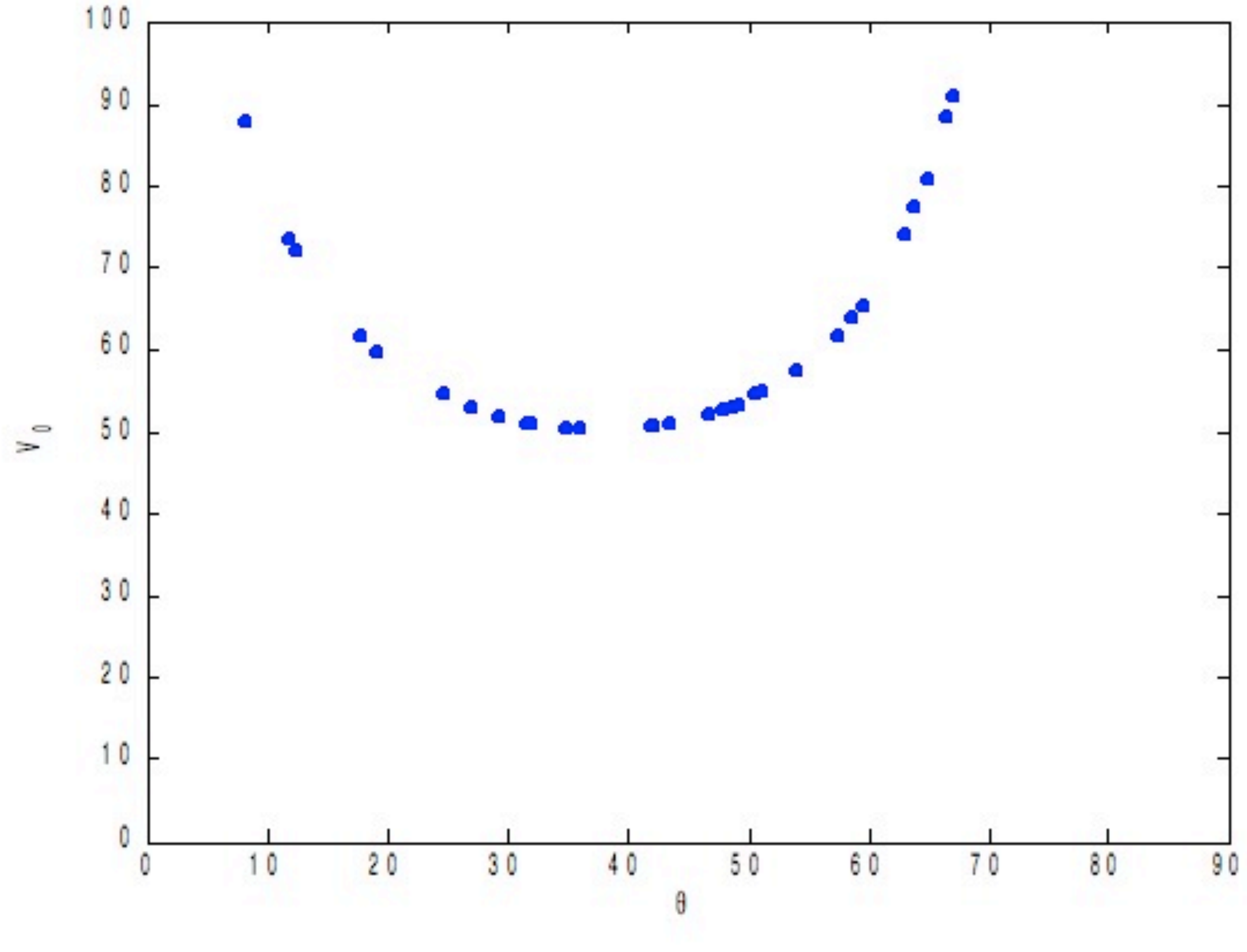


Note: multiple ways hit the target, so the solution is not unique.

Projectile continued

Many solutions!

Found by starting steepest descent at several initial conditions (multistart.m)

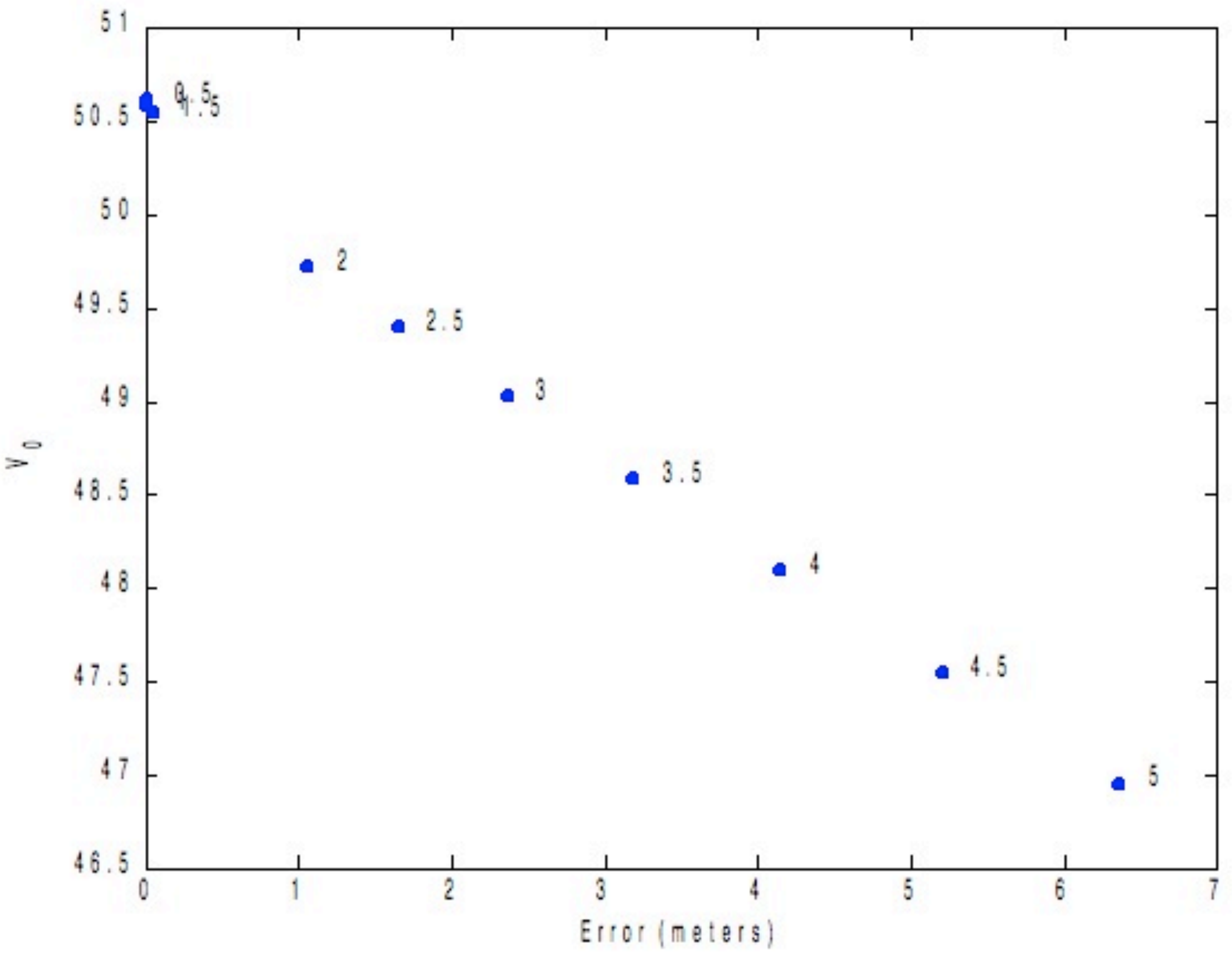


Projectile Continued

Regularization: suppose we want to use the least energy, so a lower initial velocity is better:

minimize f(v0 ,θ) + α2v0

Result (regularize.m):



Numbers on the plot indicate the α value

Choose the α that "best" solves the problem:

If we only need to be accurate to 5 meters (say), then choose α=4.5 to get the smallest initial velocity that works