Lecture 17

Optimization in one dimension

1 Introduction

Optimization is the process of finding "the best" of a set of possible alternatives. If the alternatives are described by a single continuous variable *x*, and the "goodness" of an alternative is given by the value of a function $y=f(x)$, then optimization is the process of finding the value $x=x_0$ where $f(x)$ takes on its maximum value. In many applications $f(x)$ will measure the "badness" of an alternative (error, cost, etc.) and then our goal is to find where $f(x)$ takes on its minimum value. Since finding the minimum of $g(x)$ is equivalent to finding the maximum of $-g(x)$ a simple sign change converts a minimization problem into a maximization problem and conversely. Therefore, we can focus on minimization alone with no loss of generality.

From calculus we know that at the extreme values of a continuous, differentiable function $f(x)$ the derivative $f'(x)$ is zero. This suggests that we might simply apply our root finding techniques to $f'(x)$. However, as shown in Fig. 1, $f'(x)=0$ is not a sufficient condition for a minimum. If $f''(x)<0$ the point is a maximum and if $f''(x)=0$ it may be an inflection point. To uniquely identify a minimum we must have two conditions satisfied: $f'(x)=0, f''(x)>0$. Therefore a useful algorithm must do more than just find a root of $f'(x)$. Nevertheless, as we

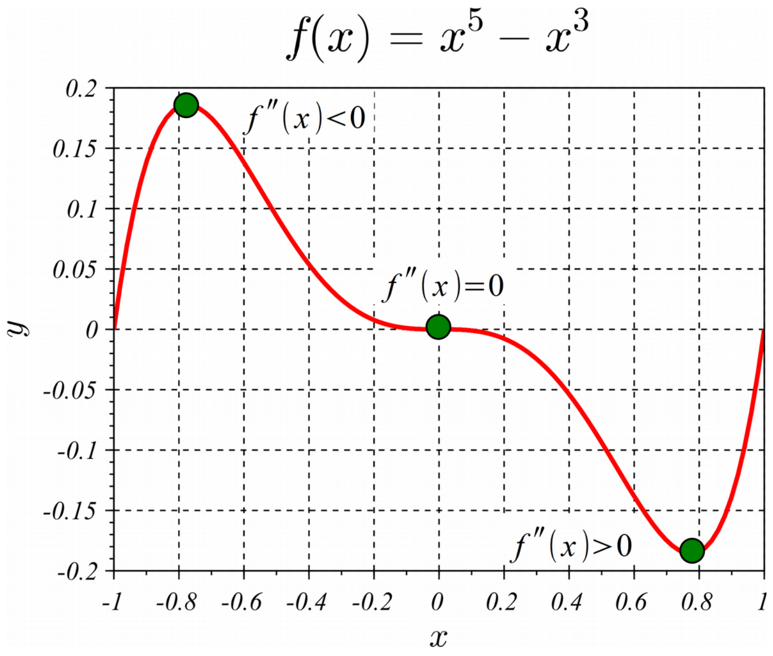

Fig. 1 The condition $f'(x)=0$ can correspond to (left point) a maximum, (middle point) an inflection point or (right point) a minimum. The sign of $f''(x)$ distinguishes these cases.

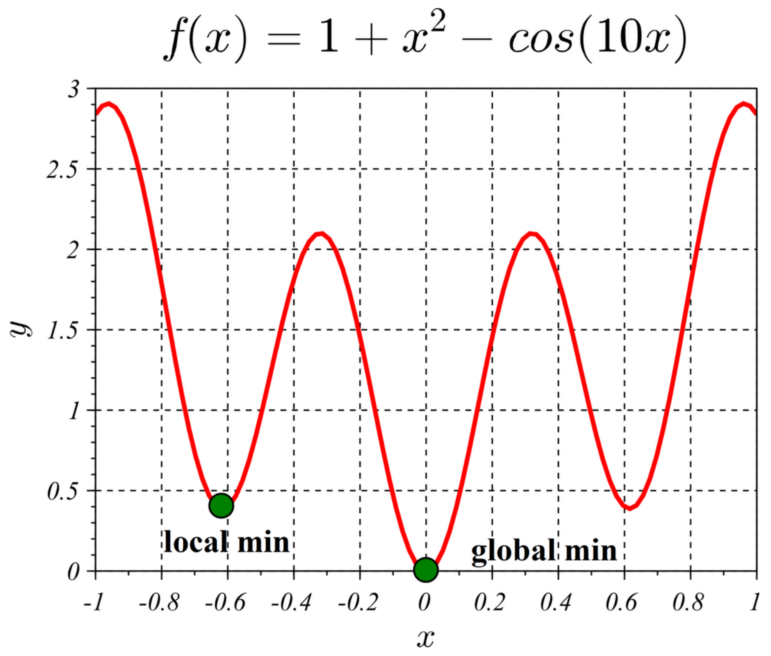

Fig. 2 Both points satisfy $f'(x)=0, f''(x)>0$ but only $x=0$ is a global minimum.

will see, there are many commonalities between optimization and root-finding algorithms.

One difficultly with optimization is illustrated in Fig. 2. The condition $f'(x)=0, f''(x)>0$ tells us only that x is a *local minimum* of $f(x)$, not if it is the *global minimum* of $f(x)$. Unfortunately there are no good, general techniques for finding a global minimum. In calculus the algorithm given for finding a global minimum is typically to first find *all* local minimum values and then identify the least of those as the global minimum. For the same reason that it is not numerically feasible to find all zeros of an arbitrary function, it is not feasible to find every local minimum of an arbitrary function. Therefore we will focus on trying to find a single local

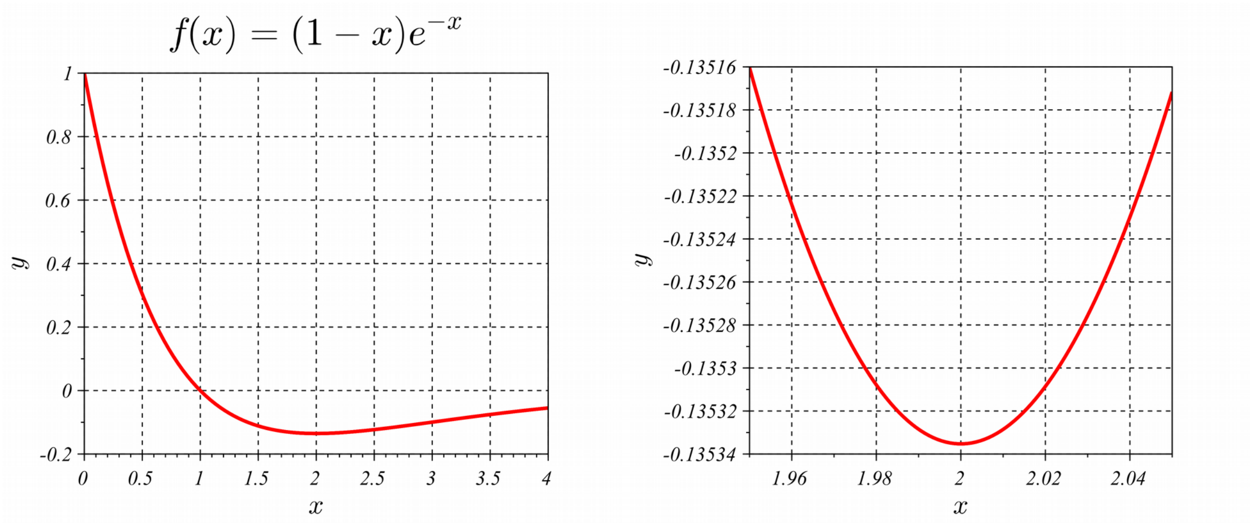

Fig. 3 Graphical method for finding a minimum.

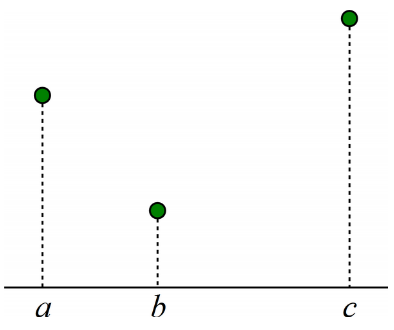

Fig. 4 Bracketing a minimum. If these are samples of a continuous function there must be a minimum in $[a,b]$.

minimum.

2 Graphical solution

Similar to root finding, simply plotting $y=f(x)$ and visually identifying the minimum is typically the easiest and most intuitive approach to optimization. This is illustrated in Fig. 3. However we often need an automated way to to optimize a function. We now turn to the optimization version of the bisection method for root find, the so-called golden search method.

3 Golden search

The slow-but-sure bisection method for root finding relies on the idea of bracketing a zero. Recall that for a continuous function, if the signs of $f(a), f(b)$ are different then there must be a zero in the interval $[a,b]$. To *bracket a minimum* we need three points $a<b<c$ (or $a>b>c$) such that $f(b)<f(a)$ and $f(b)<f(c)$, as illustrated in Fig. 4. If $f(x)$ is continuous over $[a,c]$ it cannot “go down and come back up” without passing through a minimum value of $f(x)$. There may be more than one local minimum, but there has to be at least one.

The golden search method is a way to shrink the interval $[a,c]$ while maintaining three points $a<b<c$ that bracket a minimum. When $|c-a|$ is less than some tolerance we can report our minimum as

$$x=b\pm\max(|b-a|,|b-c|)$$

The algorithm for shrinking the interval is illustrated in Fig. 5. We might expect that *b* should be the midpoint of the interval $[a,c]$. But if it was we would have to arbitrarily choose in which of the two equal-length subintervals $[a,b],[b,c]$ to sample $f(x)$. The most efficient strategy is to have $|b-a|<|c-b|$ and then sample $f(x)$ in the larger interval $[b,c]$ at

$$x=b+R(c-b)$$

where *R* is some constant. We will find either $f(x)<f(b)$ or $f(x)\geq f(b)$. Depending on which of these occur we relabel the *a,b,c* values as follows (Fig. 5)

$$\text{if } f(x)<f(b) \text{ set } a\leftarrow b, b\leftarrow x$$

$$\text{if } f(x)\geq f(b) \text{ set } a\leftarrow x, c\leftarrow a$$

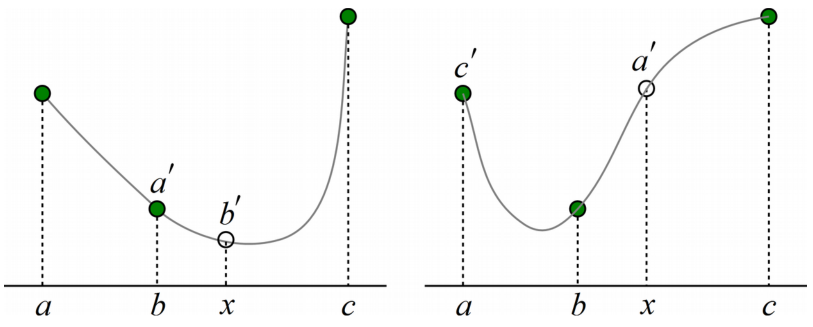

Fig. 5 left: $f(x)<f(b)$, $a\leftarrow b, b\leftarrow x$ *; right:* $f(b)<f(x)$, $a\leftarrow x, c\leftarrow a$

This gives us a new and smaller bracket. In the second case we "change direction" with *a* on the right and *c* on the left. This process is most efficient if the ratio of the resulting large and small intervals is always the same, that is

$$\frac{|c-b|}{|b-a|}=\frac{|c-x|}{|x-b|}=\frac{|b-a|}{|x-b|}$$

The value of *R* that gives this property is

$$R=\frac{3-\sqrt{5}}{2}=0.382\ldots$$

and is related to the "golden ratio" of antiquity, hence the name "golden search." We then have

$$|b-a|=R|c-a| \ \ , \ \ |x-b|=R|c-b| \ \ , \ \ |x-b|=R|x-a|$$

This algorithm converges linearly with the error decreasing by a factor of $(1-R)$ at each iteration. Function `optimGolden` in the Appendix gives a Scilab implementation of this method.

To start the golden search method we need an initial set of three bracket values *a,b,c*. A simple way to obtain these is to choose *a* near where you guess a minimum might be. Then set $b=a+h$ where *h* is some step size appropriate to your problem. If $f(b)<f(a)$ then we are "going downhill" which is what we want. If not, then swap *a* and *b* so that we are. Now take

$$c=b+(b-a)(1-R)/R\approx b+1.618(b-a)$$

If $f(c)>f(b)$ then *a,b,c* bracket a minimum. If not then set $a\leftarrow b$, $b\leftarrow c$ and continue until a bracket is found, or we give up and conclude that there is no minimum to be found. This is illustrated in Fig. 6. Function `optimBracket` in the Appendix gives a Scilab implementation of this method.

4 Parabolic interpolation

In the secant method for root finding, given two points (x_1, y_1) and (x_2, y_2) we draw a line through them to approximate the function $y=f(x)$ and find the root of that line as our next approximation to the root of $f(x)$. Given three points (x_i, y_i) , $i=1,2,3$ we can draw a unique parabola through them as an approximation of $f(x)$. We can then take the minimum of that parabola as our next approximation of the minimum of $f(x)$. This is the idea behind *parabolic interpolation*.

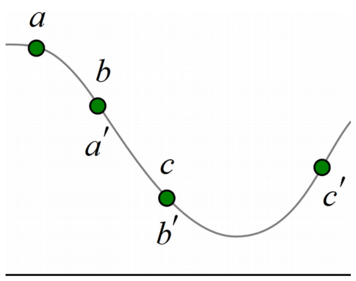

Fig. 6 Finding an initial bracket

We start with the Lagrange interpolating polynomial for our three points

$$y=y_1\frac{(x-x_2)(x-x_3)}{(x_1-x_2)(x_1-x_3)}+y_2\frac{(x-x_1)(x-x_3)}{(x_2-x_1)(x_2-x_3)}+y_3\frac{(x-x_1)(x-x_2)}{(x_3-x_1)(x_3-x_2)}$$

Now we set $dy/dx=0$. Using the fact that

$$\frac{d}{dx}(x-x_i)(x-x_j)=(x-x_i)+(x-x_j)=2x-(x_i+x_j)$$

We get

$$\frac{dy}{dx}=y_1\frac{2x-(x_2+x_3)}{(x_1-x_2)(x_1-x_3)}+y_2\frac{2x-(x_1+x_3)}{(x_2-x_1)(x_2-x_3)}+y_3\frac{2x-(x_1+x_2)}{(x_3-x_1)(x_3-x_2)}=0$$

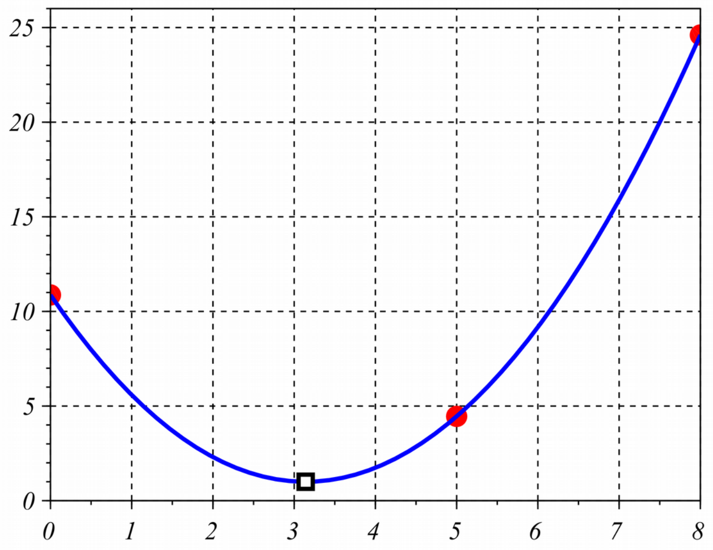

Fig. 7 Parabolic interpolation. Circles are samples. Square is minimum of interpolated parabola.

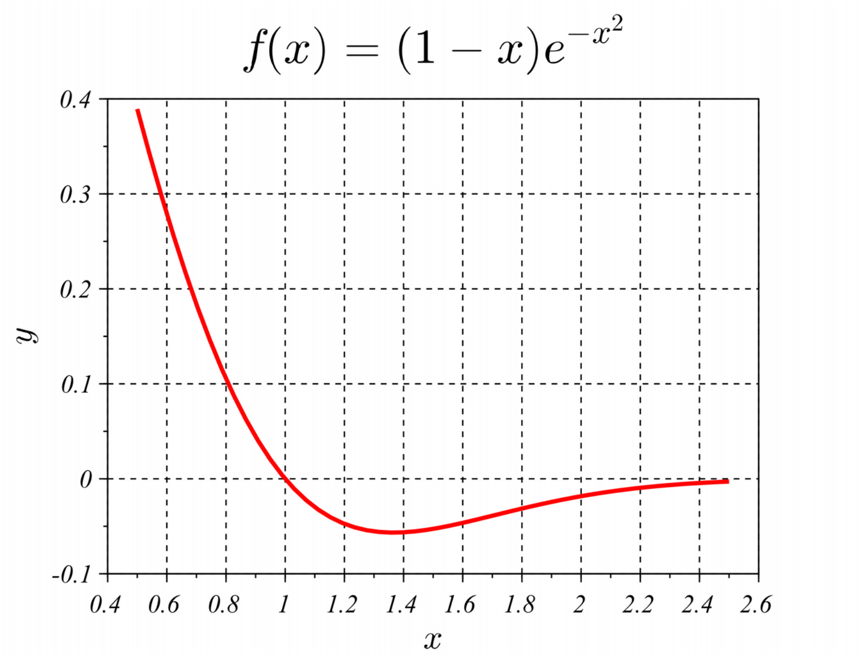

Fig. 8 Test function for minimization

Multiplying through by $(x_1-x_2)(x_1-x_3)(x_2-x_3)$ to clear fractions we find

$$y_1[2x-(x_2+x_3)](x_2-x_3)-y_2[2x-(x_1+x_3)](x_1-x_3)+y_3[2x-(x_1+x_2)](x_1-x_2)=0$$

(The minus sign for y_2 comes from $(x_1-x_2)/(x_2-x_1)=-1$.) Now we solve for *x* to find (using $(a+b)(a-b)=a^2-b^2$)

$$x=\frac{1}{2}\frac{y_1(x_2^2-x_3^2)-y_2(x_1^2-x_3^2)+y_3(x_1^2-x_2^2)}{y_1(x_2-x_3)-y_2(x_1-x_3)+y_3(x_1-x_2)}$$

Rearranging terms we obtain

$$x=\frac{1}{2}\frac{x_1^2(y_3-y_2)+x_2^2(y_1-y_3)+x_3^2(y_2-y_1)}{x_1(y_3-y_2)+x_2(y_1-y_3)+x_3(y_2-y_1)}$$

as the *x* coordinate of the minimum of the parabola.

It can be convenient to subtract x_2 from all *x* values, corresponding to a shift of the *x* axis, to obtain

$$x-x_2=\frac{1}{2}\frac{(x_1-x_2)^2(y_3-y_2)+(x_3-x_2)^2(y_2-y_1)}{(x_1-x_2)(y_3-y_2)+(x_3-x_2)(y_2-y_1)}$$

The term on the right is the displacement of the new estimate from the previous estimate x_2 . This should shrink to zero as the method converges. Parabolic interpolation converges superlinearly, $\epsilon_{k+1}\approx\alpha\epsilon_k^q$, with $q\approx 1.32$. A Scilab implementation is given in the Appendix as `optimParabolic`.

Example. The function $f(x)=(1-x)e^{-x^2}$ is plotted in Fig. 8. It has a minimum at

$$x_0=\frac{\sqrt{3}+1}{2}=1.3660254\ldots$$

Running `optimBracket` with $x_1=1.2, x_2=1.4$ identified the bracket

$$a=1.2, b=1.4, c=1.6763932$$

after one function evaluation. Running `optimGolden` with those starting values and $tol=0.0001$, 17 iterations (each involving a single function evaluation) were required to obtain

$$x_{min}=1.3660326, f(x_{min})=-0.0566378$$

Running `optimParabolic` with the same conditions, only 7 iterations were required to obtain

$$x_{min}=1.3660254, f(x_{min})=-0.0566378$$

The progress of each algorithm towards the minimum is shown in Fig. 9.

On the other hand, it is easy to find an a, b, c bracket that causes `optimParabolic` to diverge. One example is $a=1, b=2, c=3$. From Fig. 8 we can see that for $x\geq 2$, $f''(x)<0$, so we might expect problems when using a method which models the function as an upwardly curved parabola.

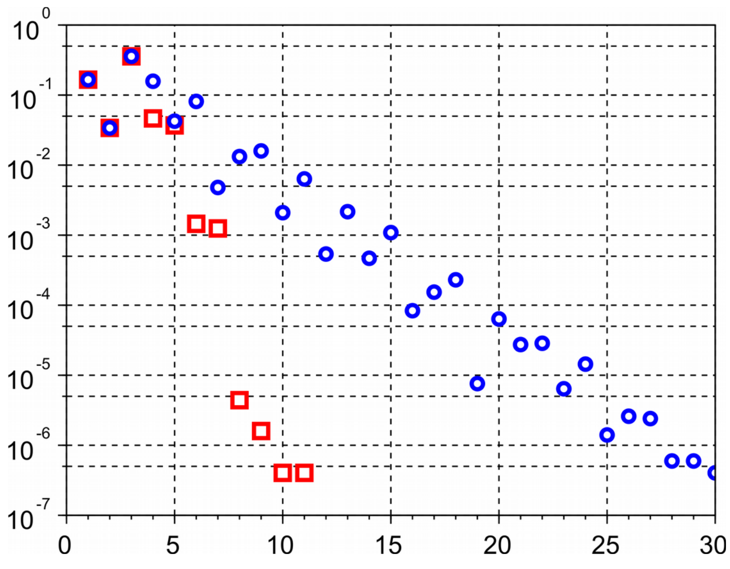

Fig. 9: Progression of golden search (circles) and parabolic interpolation (squares) toward minimization of $f(x)=(1-x)e^{-x^2}$. Horizontal axis increments at each function call. Vertical axis is $|x-r|$ on log scale. First three calls are for bracketing routine.

5 Newton's method

For root finding we saw that Newton's method gave quadratic convergence at the price of having to explicitly calculate the derivative $f'(x)$. We can apply Newton's method to solve $f'(x)=0$. We find the root of the first-order Taylor series

$$f'(x_k+h)\approx f'(x_k)+f''(x_k)h=0$$

to be

$$h=-\frac{f'(x_k)}{f''(x_k)}$$

This gives us the iteration formula

$$x_{k+1}=x_k-\frac{f'(x_k)}{f''(x_k)}$$

To apply this we need to explicitly calculate both first and second derivatives. Essentially what we are doing is to approximate the function by the second-order Taylor series

$$f(x_k+h)=f(x_{k+1})\approx f(x_k)+f'(x_k)h+\frac{1}{2}f''(x_k)h^2$$

and setting h to correspond to the minimum of this parabola. Of course this requires that $f''(x_k)>0$ otherwise the parabola will have a maximum, not a minimum. This should be the case, provided we start "close enough" to the actual minimum of $f(x)$. We could avoid explicitly calculating the derivatives by approximating $f''(x)$ (and possibly $f'(x)$ also) using function values alone, analogous to what we did in the tangent method. The result would be a *quasi-Newton method*. These often form a primary component of a state-of-the-art optimization routine.

6 Hybrid methods

As with root finding, optimization presents us a tradeoff. Sure-fire methods (golden search) are relatively slow while faster methods can be unstable and fail to find a solution. For particular functions one or the other may be preferable. For a general-purpose optimization routine, a good strategy is to combine slow-but-sure and fast-but-unstable methods into a *hybrid method*. *Brent's method* [1] is a good example. This algorithm first attempts to use parabolic interpolation, but includes tests to indicate if this is converging in a desirable manner. If it isn't, the algorithm falls back on the golden search for one or more iterations before trying parabolic interpolation again. Other hybrid algorithms employ quasi-Newton methods in an attempt to achieve rapid convergence when possible and slow-but-guaranteed convergence otherwise. These include the built-in optimization routines in Scilab and Matlab

7 The optim (Scilab) and fminunc (Matlab) functions

Scilab provides a built-in optimization routine `optim`. It will attempt to minimize a function of any number of variables. By default it expects the function being optimized to return both function and derivative values. However, numerical derivatives can be used instead. In this case the basic calling syntax is

```
[fopt,xopt] = optim(list(NDcost,f),x0);
```

Here `f(x)` is the function to be minimized, `x0` is an initial guess at the minimum, `xopt` is the computed minimum and `fopt` is the minimum function value. The `list(NDcost,f)` statement takes care of providing numerical derivatives.

Matlab provides the function `fminunc` for optimization. Its basic syntax is

```
[xopt,fopt] = fminunc(f,x0);
```

As always, there are many options that can be set, and the help browser provides complete documentation.

8 Constrained optimization

What we have covered so far is more precisely referred to as "unconstrained optimization." We are free to test any values of x in our search for a minimum. In a *constrained optimization* problem only x values that satisfy one or more given constraints are valid candidates for a minimum. A simple example would be

$$\min(x+1)^2 \text{ s.t. } x\geq 0$$

Here we want to find the minimum of $f(x)=(x+1)^2$ but subject to the constraint that x is non-

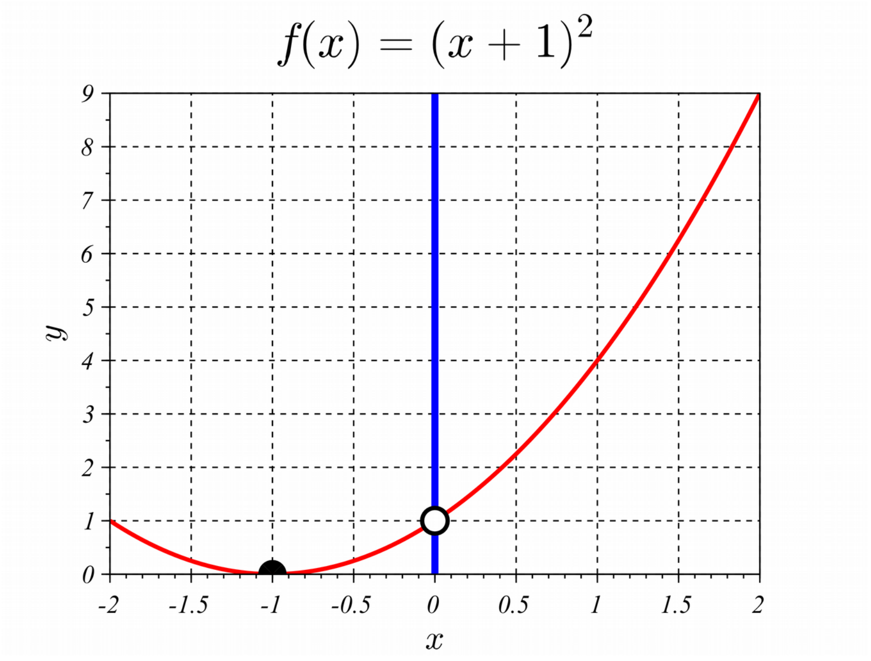

Fig. 10 Minimization without constraint (solid dot) and with constraint $x\geq 0$ (open dot).

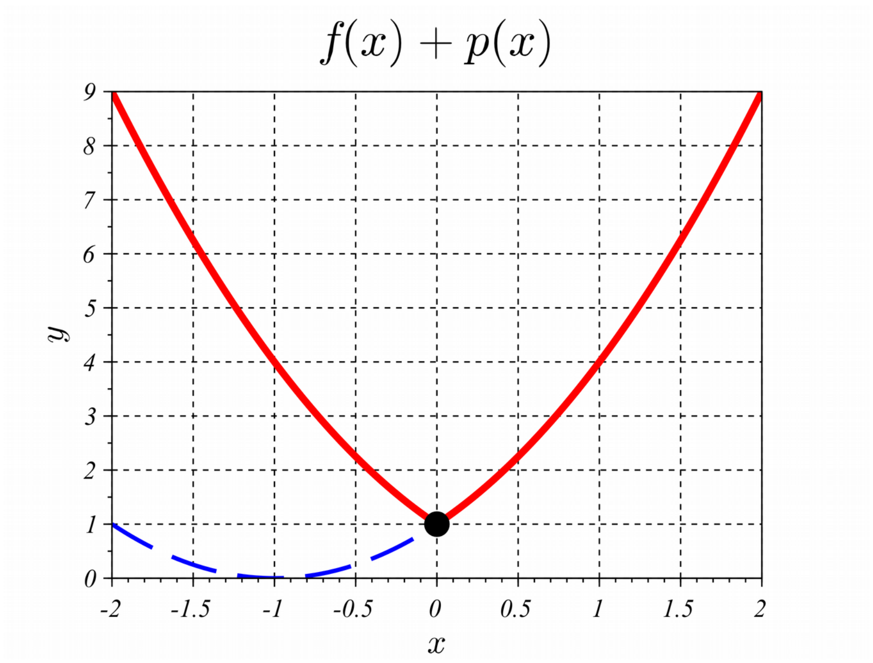

Fig. 11 Constrained optimization using a penalty function.

negative. This is illustrated in Fig. 10. With no constraint our solution would simply be the bottom of the parabola. But with the constraint the best we can do is $x=0$. Implementing constrained optimization can be tricky, depending on the complexity of the constraints. A simple work around that allows us to implement contained optimization using unconstrained optimization algorithms employs the idea of a *penalty function*. Instead of minimizing $f(x)$, we minimize $f(x)+p(x)$ where the penalty function $p(x)$ is large for values of *x* that violate the constraints and zero otherwise. For example, adding

$$p(x)=\begin{cases} 0 & ,x\geq 0 \\ -4x & ,x<0 \end{cases}$$

to $f(x)=(x+1)^2$ results in the function shown in Fig. 11. Now an unconstrained minimization of $f(x)+p(x)$ produces the same solution as the original constrained minimization problem.

9 References

1. Brent, Richard P. *Algorithms for Minimization Without Derivatives*. Dover Publications. Kindle edition. ASIN: B00CRW5ZTK. 2013 (Originally published 1973)

10 Appendix – Scilab code

10.1 Golden search

```
//////////////////////////////////////////////////////////////////////
// optimGolden.sci
// 2014-10-31, Scott Hudson, for pedagogic purposes.
// Implements golden search for minimization of y=f(x).
// (a,b,c) must bracket a minimum, i.e., f(b)<f(a) & f(b)<f(c)
// with a<b<c or a>b>c. Function terminates when minimum has been
// estimated to within +-tol
//////////////////////////////////////////////////////////////////////
function [xmin, fmin]=optimGolden(a, b, c, fa, fb, fc, f, tol)
  R = (3-sqrt(5))/2; //golden ratio
  while (abs(c-b)>tol) //abs(c-b)>abs(b-a) is upper bound on error
    x = b+R*(c-b);
    fx = f(x);
    if (fx<fb)
      a = b;
      fa = fb;
      b = x;
      fb = fx;
    else
      c = a;
      fc = fa;
      a = x;
      fa = fx;
    end
  end
  xmin = b;
  fmin = fb;
endfunction
```

10.2 Bracketing a minimum

```
//////////////////////////////////////////////////////////////////////
// optimBracket.sci
// 2014-10-31, Scott Hudson, for pedagogic purposes.
// Given intial values x1,x2 and a function y=f(x), attempts to
// follow the function downhill until a minimum has been bracketed
// f(b)<f(a) & f(b)<f(c) with a<b<c or a>b>c.
//////////////////////////////////////////////////////////////////////
function [a, b, c, fa, fb, fc]=optimBracket(x1, x2, f)
  MAX_ITERS = 20; //give up after this many attempts
  a = x1;
  b = x2;
  fa = f(a);
  fb = f(b);
  if (fa<fb) //going uphill, go other way by switching a & b
    c = a;    //save a
    fc = fa;
    a = b;    //a<-b
    fa = fb;
    b = c;    //b<=a
    fb = fc;
  end
  R = (3-sqrt(5))/2; //golden ratio
  step = (1-R)/R;
  done = 0;
  iter = 0;
  while (~done)
    c = b+(b-a)*step;
    fc = f(c);
    if (fc>fb)    //we're now going uphill, bracket found
      done = 1;
    else          //still going down hill
      a = b;
      fa = fb;
      b = c;
      fb = fc;
      iter = iter+1;
    end
    if (iter>MAX_ITERS)
      error('optimBracket: MAX_ITERS reached');
    end
  end
endfunction
```

10.3 Parabolic interpolation

```
//////////////////////////////////////////////////////////////////////
// optimParabolic.sci
// 2014-10-31, Scott Hudson, for pedagogic purposes.
// Uses parabolic interpolation to estimate the minimum of y=f(x).
// Last three estimates are retained for interpolation.
//////////////////////////////////////////////////////////////////////
function [xmin, fmin]=optimParabolic(a, b, c, fa, fb, fc, f, tol)
  MAX_ITERS = 20;
  x = [a,b,c];
  y = [fa,fb,fc];
  iter = 1;
  while ((max(x)-min(x))>2*tol)
    N = (y(3)-y(2))*x(1)^2+(y(1)-y(3))*x(2)^2+(y(2)-y(1))*x(3)^2;
    D = (y(3)-y(2))*x(1)  +(y(1)-y(3))*x(2)  +(y(2)-y(1))*x(3)  ;
    x(1) = x(2);
    y(1) = y(2);
    x(2) = x(3);
    y(2) = y(3);
    x(3) = N/(2*D);
    y(3) = f(x(3));
    iter = iter+1;
    if (iter>MAX_ITERS)
      error('optimParabolic: MAX_ITERS reached');
    end
  end
  xmin = x(3);
  fmin = y(3);
endfunction
```