Optimization

2018

**Book:**

Numerical Optimization, Second Edition (2006), by Jorge Nocedal, S. Wright

# Fundamentals of Unconstraint Optimization

## What is a Solution?

**Taylor's Theorem**

The approximation / model used in optimization is:

The and are all evaluated at .

More formally, for some :

**Regarding the in**

Suppose we have *x* near value *a*:

Taking second derivative of both sides, we have:

**The expansion for**

Example: means the second derivative effects are modeled by .

**Necessary and Sufficient Conditions for a Local Minimum**

First order necessary condition: .

Second order necessary condition: and is positive semidefinite.

A positive semidefinite matrix *M* means that for all .

Second order sufficient conditions: and is positive definite.

The second-order sufficient conditions are not necessary. For example, in the second derivative is 0 at , so we don't have , but that point is the minimum.

## Overview of Algorithms

**Steepest Descent Justification**

Use the approximation:

Suppose we are currently at point , the direction that produce the greatest decrease needs to minimize the term .

The dot product is minimum when cos(θ) = -1. So points in opposite direction to .

**Newton Direction (Step)**

For a function we are at and the second order Taylor approximation is

Simplifying the notation and calling this second order model *mk(p)*:

Suppose the is positive definite, then this is like a parabola facing upward and there is a minimum at where .

To take the derivative of , we need the following results:

The derivative matrix is symmetric, so

The derivative is zero at:

This *p* is the **Newton direction**.

*Unlike this is actually a step, not simply a direction. It's actually how far you need to travel, so to make the (second) derivative zero, hence arriving at the minimum.*

Numerical example:

Suppose we are at (x, y) = (2, 3)

So makes the jump straight to the minimum, which is at (0,0).

**The Impact of being Positive Definite**

*Reminder: if A is positive definite, then A-1 is also positive definite.*

For the Newton direction vector *p* found by , we have:

A positive definite matrix *A* means *xTAx* is positive for all *x*. This means .

*Earlier on, the first order part of the model is . The is applying the first order model, using as the step.*

*So, this makes "p" a descent direction. But this is to be expected - "p" is more than the descent direction, it goes straight to the second order model's minimum.*

**Quasi-Newton Methods**

Instead of the true Hessian , the quasi-Newton methods use an approximation *Bk* that obeys the secant equation:

The Hessian approximation is being inferred from the change in the gradient.

**Scaling**

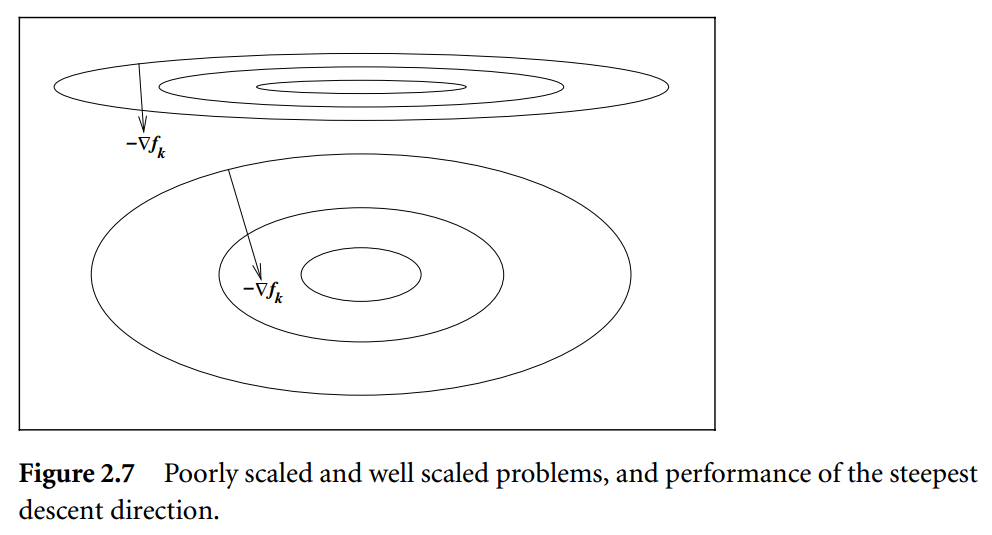
A problem is **poorly scaled** if changes to one variable produce a much larger output change than changes made to the other variables.

For example, is a poorly scaled problem because the output change is dominated by the *x1* term.

The steepest descent does not work well for poorly scaled problems.

As an example, consider:

Pictorially the concept is



The steepest descent does not work well for *f1(x1,x2)*, the top picture, since the negative gradient vector is pointing far away from the center.

Mathematically for , decreasing the variable tends to have more impact in decreasing *f1*, so the algorithm tends to favor decreasing *x2*.

# Line Search Methods

## Step Length

**Function of Step Length φ(α)**

In the line search method, .

We would like to choose *α* so to produce the greatest amount of decrease possible.

The choice of *α* can be viewed as an optimization problem, where

and we are trying to minimize φ() by varying α.

**The Wolfe Conditions for choosing α**

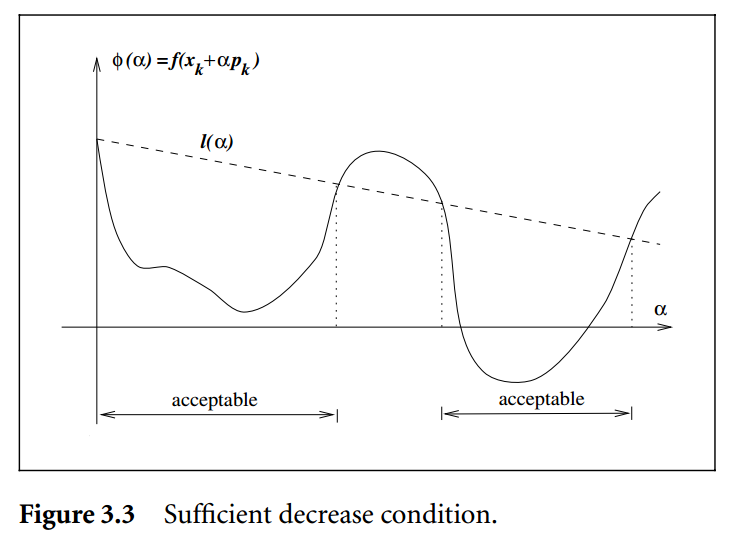
|  |  |
| --- | --- |
|  | (3.6a) |
|  | (3.6b) |

with . The *c1* is quite small, say *c1 = 10-4*.

The first condition requires a minimum decrease.

The is the derivative in the direction of .

The right hand side is a line that slopes downward, but not as steep as the tangent.

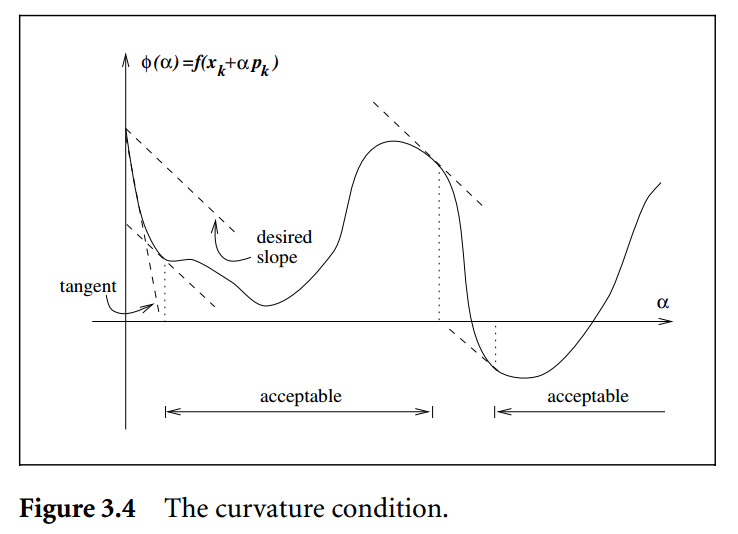


The second condition require the slope at the future point be not too negative.

The left hand side is the slope at the future point. We are minimizing so all slopes are negative.

Suppose the current slope is -5, and *c2 = 2.5*. We want (future slope) > -2.5.

The idea is that if the future slope is say -3, then it's still quite negative, and we haven't travel far enough.



**Strong Wolfe Conditions**

Previously the second wolf condition says

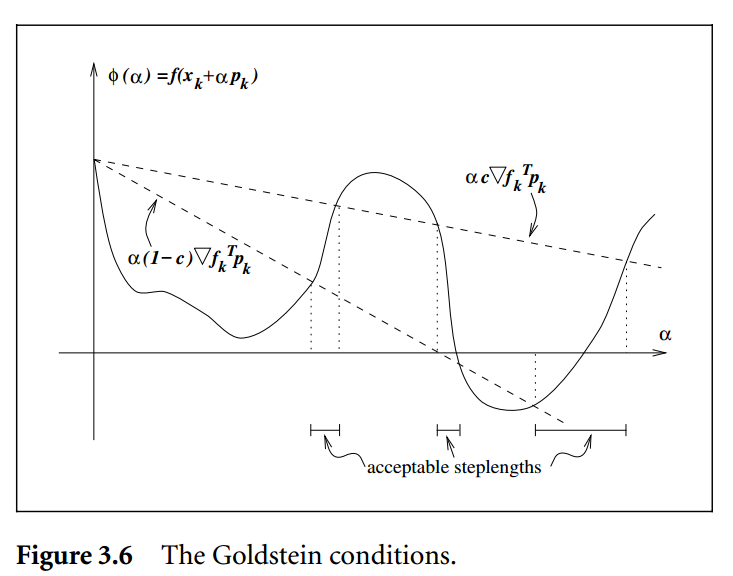
So if the future slope is some positive number, then it's still okay. But that means the future point has overshot the minimum. The strong Wolfe conditions require

This rules out the future point from having a large positive slope.

**The Goldstein Conditions**

with .

The upper and lower bounds represented by this requirement are shown in the following figure:



## Step-Length Selection Algorithms

**Convex Quadratic Minimizer**

Let be convex quadratic, so that a minimum actually exists.

The minimizer along the ray is

|  |  |
| --- | --- |
|  | (3.55) |

Derivation:

Let

This is because the directional derivative is the dot product of the gradient and the direction.

Set , solve for α. Again, this is only possible due to the *f(x)* being convex quadratic.

## Gradient Descent in Python

**Summary**

The following demo uses gradient descent to solve the basic least square problem .

Given a linear system , the function to minimize is .

For details on how this gradient is derived, see the notes for "Numerical Algorithms", CH 4.

**Simple gradient descent has very poor convergence**

The result of the code shows that gradient descent has a hard time converging, even in the best scenario, I wrote:

run\_test(x1=100, x0=100, data\_low=0, data\_high=1)   # usually works

I was surprised to see that even in this case, when the gradient components should be balanced, it does not converge all the time.

When gradient components are not balanced, the simple gradient descent implemented here often do not fully converge. It will only converge for the component that has a large gradient value. The step size then drops to very low due to overshooting the large gradient component, and the smaller gradient component will not converge.

Even when gradient descent does not converge, the first few iterations do lower the loss substantially.

**Call tree**

run\_test(data\_low, data\_high, x1, x0)  
 |🡪 generate\_data(data\_low, data\_high, x1, x0)  
 |🡪 gradient\_descent(A, b)  
 | |🡪 gradient(x, A, AT\_b)  
 |

|🡪 print\_info(str\_list)

**Code**

import numpy, random

# print data in columns

def print\_info(str\_list):

    for s in str\_list:

        print(s.center(15), end='')

    print()

# generate data

def generate\_data(data\_low, data\_high, x1, x0):

    x\_data = []

    y\_data = []

    for i in range(10):

        #############################

        x = random.uniform(data\_low, data\_high)

        noise = random.gauss(0, 0.1)

        y = x1\*x + x0 + noise

        #############################

        x\_data.append(x)

        y\_data.append(y)

    return x\_data, y\_data

# gradient computation

def gradient(x, A, AT\_b):

    return A.T.dot(A.dot(x)) - AT\_b

# gradient descent

def gradient\_descent(A, b):

    AT\_b = A.T.dot(b)

    x = numpy.array([[0, 0]]).T

    step = 1

    print\_info(["loss", "step", "x[0]", "x[1]"])

    for i in range(0, 50):

        old\_loss = numpy.sum(numpy.abs(A.dot(x) - b))

        x\_old = x.copy()

        x = x - step \* gradient(x, A, AT\_b)

        loss = numpy.sum(numpy.abs(A.dot(x) - b))

        while loss > old\_loss:

            x = x\_old

            step = step / 2

            x = x - step \* gradient(x, A, AT\_b)

            loss = numpy.sum(numpy.abs(A.dot(x) - b))

            if step < 1e-8: return

        print\_info(["{0:.3g}".format(loss),

                    "{0:.3g}".format(step),

                    "{0:.3g}".format(x[0][0]),

                    "{0:.3g}".format(x[1][0])])

def run\_test(data\_low, data\_high, x1, x0):

    # generate data

    (x\_data, y\_data) = generate\_data(data\_low, data\_high, x1, x0)

    # Ax = b format

    A = numpy.array([x\_data,

                     numpy.ones(len(x\_data))], dtype=numpy.double).T;

    b = numpy.array([y\_data]).T

    # solve using gradient descent

    gradient\_descent(A, b)

    # solve using normal equations

    print("Normal equation solution:")

    print(numpy.linalg.solve(A.T.dot(A), A.T.dot(b)))

    print()

run\_test(x1=100, x0=100, data\_low=0, data\_high=1)   # usually works

run\_test(x1=100, x0=100, data\_low=0, data\_high=10)  # cannot converge

run\_test(x1=1000, x0=10, data\_low=0, data\_high=1)   # sometimes converge very slowly

                                                    # occasionally cannot converge

run\_test(x1=10, x0=1000, data\_low=0, data\_high=1)   # sometimes converge very slowly

                                                    # occasionally cannot converge

# Trust-Region Methods

**Strategy for Choosing the Trust Region Radius (Δk)**

Compute a metric of how good the model is:

The numerator is the actual reduction when moving from to . The denominator is the theoretical reduction.

If this ratio is:

negative --- then the model is wrong. The model is predicting a reduction whereas the numerator term saw an increase. This should not happen as the step should be rejected in the first place.

close to one --- the model is good and the trust region radius (Δk) can be expanded for the next iteration

positive but smaller than one --- no change to the trust region

significantly smaller than one --- shrink the trust region

## Algorithms Based on the Cauchy Point

**The Dogleg Method**

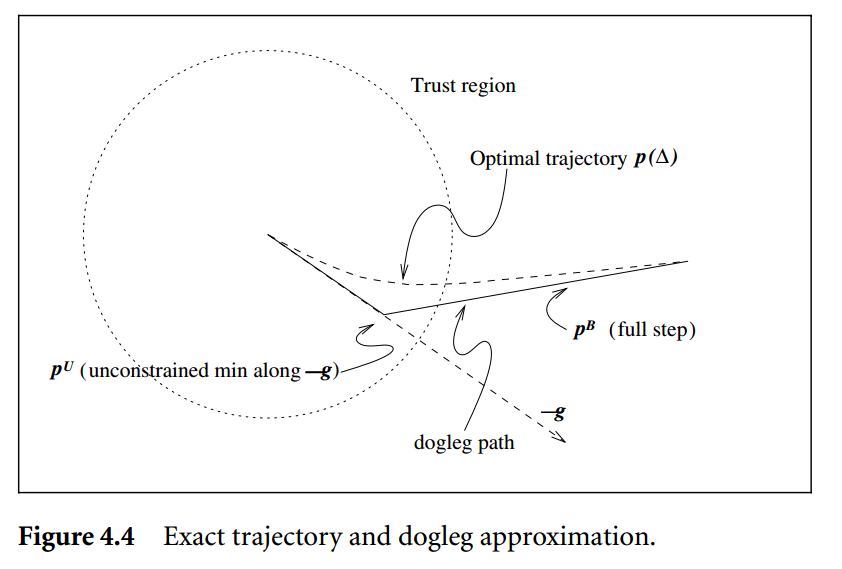
dogleg definition --- a bend, like how a dog's hid leg bends.

This method assumes that the model is convex quadratic:

with the minimum at:

If this *pB* is inside the trust region radius Δ, just head straight to it.

If the *pB* is outside of the trust region, first head along the steepest descent path, and then head toward *pB*.



The distance to travel along the steepest descent *g* has been computed in the section "Convex Quadratic Minimizer" as

where the *p* is the direction to travel.

In this case, the travelling is done in the direction of the gradient *-g*, called *pU* in the figure.

with

The second leg of the journey runs between *pU* and *pB*.

Both legs can be described parametrically:

We can only travel as far as the trust region radius Δ allow.

# Conjugate Gradient Methods

## The Linear Conjugate Gradient Method

**as a minimization problem**

This algorithm assumes that A is symmetric positive definite. Therefore the is an upward facing parabola that has only one minimum.

At the minimum, φ(x)=0 and so Ax=b.

**conjugacy**

A set of vectors is said to be **conjugate** with respect to matrix *A* if:

|  |  |
| --- | --- |
|  | (5.5) |

**conjugate vectors are linearly independent**

Start with

Multiply both sides by

Only the first term is non-zero, all the other terms are zero due to the definition of conjugacy.

, due to the positive definite requirement, so c0=0.

This argument can be repeated for other terms to show that the coefficients c0 through cn-1 are all zeros.

**Optimizing along conjugate directions converge to the minimum in at most *n* steps (Theorem 5.1)**

Suppose we generate the sequence {xk} using:

where the αk comes from the optimal step size (equation 3.55):

The sequence {xk} converges to the minimum in at most *n* steps.

Proof:

The conjugate direction vectors {*pi*} are linearly independent, so the minimum x\* can be described as:

The goal is to show that these "σ" values are actually the same as the "α" values.

Multiply both sides by , for a particular *k* between 0 and *n-1*:

On the right hand side most terms will go to zero due to the definition of conjugacy.

Separately:

Multiply both sides by and most of the terms on the right will be zero, leaving:

So it's okay to replace with .

The , so the σk is the same thing as the αk.

Note that does not mean . The x vector is n x 1, while the final result is a scalar. So there is an n to 1 relationship, and it's possible for different vectors to map to the same scalar answer. The cancellation law is only valid if there is a 1 to 1 relationship.

**Optimizing along conjugate directions amounts to minimization of an expanding subspace (Theorem 5.2)**

|  |  |
| --- | --- |
|  | (5.11) |

This implies that is the minimizer of over the set .

Proof

For the case k=1, . The is minimum along the direction due to the selection of α0 according to equation (3.55).

Being the minimum along implies .

For cases k=2,3, ..., induction is used. The goal is to first show , then the minimization criteria follows naturally.

Separately, starting with:

leads to:

|  |  |
| --- | --- |
|  | (5.10) |

For the case :

This term is zero due to the definition of αk being:

For the cases :

This expression is also zero. The first term is zero due to the induction hypothesis. The second term is zero due to the definition of conjugate vectors.

**The Conjugate Gradient Algorithm, Preliminary Version (Algorithm 5.1)**

Previously, the conjugate vectors are simply given.

It turned out that conjugate vectors can be calculated using:

|  |  |
| --- | --- |
|  | (5.14e) |

To find the scalar value , multiply both sides by :

|  |  |
| --- | --- |
|  | (5.14d) |

Preliminary version of the conjugate gradient algorithm:

**Properties of the Conjugate Gradient Algorithm (Theorem 5.3)**

Each search direction and residual is contained in the Krylov subspace:

Proof

The proof is by induction.

If these statements are true for *k*, then:

Equation (5.10):

The basis of the induction assumes is in span of

Most of the right-hand side vectors are already part of . The additional basis being brought in here is .

Therefore, the conclusion is .

Equation (5.14e):

A previous result is:

Again, most of these vectors are already part of . The additional basis being brought in here is .

Therefore, the conclusion is .

The successive vectors are conjugate by construction --- see equation (5.14d) and how the constant is tuned to make successive vectors conjugate.

On the inductive step, we are on index *k+1*. The *i=k* case is always true by construction. What's needed is to prove the *i=k-1, k-2, ...* cases.

Equation (5.14e):

This is the inductive step, so we are on index *k+1*. The statement is assumed to be true for index *k*. The second term on the right side is therefore zero.

A previous part of this theorem: .

Equation (5.11), Theorem 5.2:

We are only considering up to *i=k-1*, therefore the highest *i* being considered is:

This is within the range of theorem 5.2, so

Unlike the previous proofs, this one is direct.

Equation (5.14e):

Equation (5.11), Theorem 5.2: .

Therefore, .

**Practical Form of the Conjugate Gradient Method (Algorithm 5.2)**

The modified formulas are in green.

|  |  |
| --- | --- |
|  | (5.24a) |

This is equation (5.10). The original definition is , while equation (5.10) is faster to compute due to already computed in equation (5.24a).

|  |  |
| --- | --- |
|  | (5.24d) |

Proofs:

This algorithm uses optimized formulas for αk and βk+1.

The new formula for α:

The old formula is equation (3.55):

The new formula is changing the numerator term to for faster computation - because this term also appears in the new formula for β.

Equation (5.14e):

Equation (5.11): .

So the second term is zero, and .

The new formula for β:

The old formula is equation (5.14d):

Numerator changes use equation (5.10):

Denominator changes use the new α formula, which is equation (5.24a).

Plugging the new numerator and denominators into the old β formula:

Theorem 5.3:

So .

## Linear Conjugate Gradient in Python

**Summary**

The conjugate gradient algorithm is from the book's "Algorithm 5.2".

The following example solves the least square problem by applying the conjugate gradient algorithm to the normal equation.

Convergence is very fast. In theory, for "*n*" variables, conjugate gradient converges in "*n*" iterations. However, in the tests done, it seems to converge in just 1% of "n" operations - so taking just around 30 iterations to solve a least squares problem with 3000 variables.

**Code**

import numpy, random

import matplotlib.pyplot as pyplot

def generate\_data(data\_low, data\_high, x\_low, x\_high):

    """Returns A\_data, b\_data, x"""

    num\_data = 12000

    num\_vars = 3000

    A\_data = numpy.random.uniform(data\_low, data\_high, (num\_data, num\_vars))

    x = numpy.random.uniform(x\_low, x\_high, (num\_vars, 1))

    b\_data = A\_data.dot(x) + numpy.random.normal(0, 0.1, (num\_data, 1))

    return A\_data, b\_data, x

def conjugate\_gradient(A, b):

    """Returns loss\_list, x"""

    length = b.shape[0]

    x = numpy.zeros((length, 1))

    r = A.dot(x) - b

    p = -1 \* r

    loss\_list = []

    loss\_list.append(numpy.sum(numpy.abs(r)))

    r\_dot\_r = r.T.dot(r)[0][0]

    # allow error of 0.05 per equation

    stop\_condition = 0.05 \* length

    for i in range(0, length):

        A\_times\_p = A.dot(p)

        alpha = r\_dot\_r / (p.T.dot(A\_times\_p))[0][0]

        x = x + alpha \* p

        r\_next = r + alpha \* A\_times\_p

        r\_next\_dot\_r\_next = r\_next.T.dot(r\_next)[0][0]

        beta = r\_next\_dot\_r\_next / r\_dot\_r

        p = -1 \* r\_next + beta \* p

        r = r\_next

        r\_dot\_r = r\_next\_dot\_r\_next

        loss = numpy.sum(numpy.abs(r))

        loss\_list.append(loss)

        if loss < stop\_condition: break

    return loss\_list, x

def run\_test(data\_low, data\_high, x\_low, x\_high):

    A\_data, b\_data, x\_real = generate\_data(data\_low, data\_high, x\_low, x\_high)

    # Get into conjugate gradient's "Ax = b" form

    A = A\_data.T.dot(A\_data)

    b = A\_data.T.dot(b\_data)

    loss\_list, x = conjugate\_gradient(A, b)

    # check "x" against "x\_real"

    print("sum(|x - x\_real|) =", numpy.sum(numpy.abs(x\_real - x)))

    loss\_list = numpy.log10(loss\_list)

    # plot the "loss\_list"

    pyplot.plot(loss\_list, "bo")

    pyplot.ylabel("log10(loss)")

    pyplot.xlabel("Iteration")

    pyplot.show()

run\_test(data\_low=0, data\_high=100, x\_low=0, x\_high=20)     # loss sometimes increase

run\_test(data\_low=0, data\_high=1000, x\_low=0, x\_high=200)   # loss sometimes increase

run\_test(data\_low=-100, data\_high=100, x\_low=0, x\_high=200)

# Quasi-Newton Methods

## The BFGS Method

**Quadratic Modeling (review)**

The model is:

where *B* is symmetric positive definite

The minimum is at:

The update rule is:

**Criteria for Hessian Approximation - the Secant Equation**

Suppose we are at iteration *k*. The next model *mk+1* will be:

We would like the model (*mk+1*) to correctly produce gradients at two points: *xk+1* and *xk*.

For model *mk+1* to correctly produce the gradient at *xk+1*:

which is automatically true since the model *mk+1* will actually use the gradient at *xk+1*.

The real constraint is to produce the correct gradient at the point *xk*:

Evaluate the gradient for model *mk+1*:

Let:

to get , the secant equation.

**The Curvature Condition**

When *B* is positive definite,

Reason:

**The Wolfe condition leads to the curvature condition**

The secant equation defines:

So the curvature condition is:

Separately, the second Wolfe condition, equation (3.6b):

In the current context:

Applying the Wolfe condition:

The Wolfe condition uses , so the term is negative.

The term is negative as well due to the Wolfe condition requiring there be a decrease. So the conclusion is that is positive.

**The BFGS update formula produces positive definite matrices**

The update formula is:

The *Hk+1* will always be positive definite as long as *Hk* is also positive definite.

Discussion:

The "" term is positive as long as *Hk* is positive definite.

The second term on the right:

The *r­k* is positive due to the curvature condition .

**The initial estimation for H**

The initial estimation is a scaled identity matrix.

Explanation of the scaling constant:

The purpose of *H* is , meaning to predict given .

However, in the very beginning, *H* is an identity matrix, so that is always in the direction of . The closest prediction possible is made by the projection of onto .

# Large-Scale Unconstrained Optimization

## Limited-Memory BFGS

This is also known as the L-BFGS.

The goal is to avoid storing the *H* matrix in memory. The idea is to keep just the last few and vectors. The *H* matrix is then "rebuilt" from the identity matrix for each iteration.

The BFGS update formulas:

Example --- if we keep just the last *m=3* vectors, and are currently on the *k=100* iteration:

Evaluating requires a call to . So this algorithm is recursive.

When the evaluation happens from right to left, and there is no matrix multiplication.

For example, the term , if evaluated from left to right, will form a matrix due to the . But if is done first, then no matrix is formed.

# Calculating Derivatives

**Forward-difference, or one-sided-difference, approximation**

**Central difference formula**

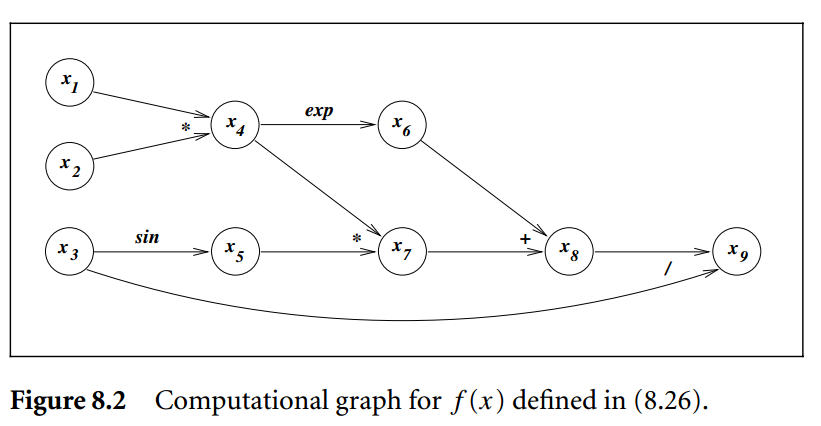
This is more accurate than the forward-difference approximation. If *f(x)* is increasing, then the forward difference is always going to over-estimate the slope.

However it is also about twice as expensive --- the term in the forward-difference equation is the same for the different *xi* variables.

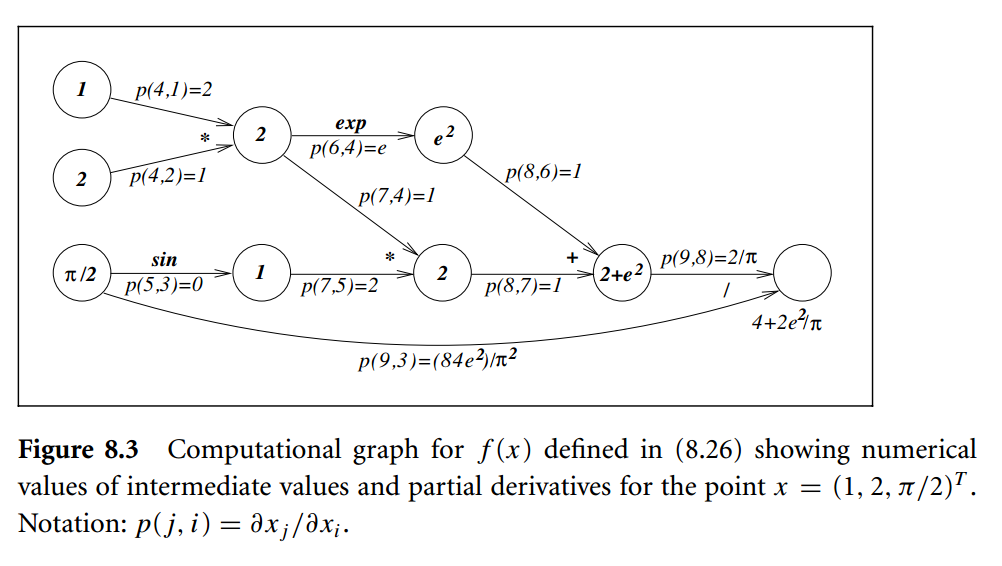
## Automatic Differentiation

**The Example**

**The computation graph**



**Node values and derivative values**



There are three mistakes in this figure and they have been edited in red.

The value for each node is computed first. For example, the node circled in green is *x4*:

After all the node values are computed, the partial derivatives, which are the edges, are computed. For example, :

Now the term *x8* actually contains *x­3* inside of it, that is, *x8* depends on *x3*. But for the purpose of this algorithm, the *x­8* is treated as a variable independent of *x3*. We write

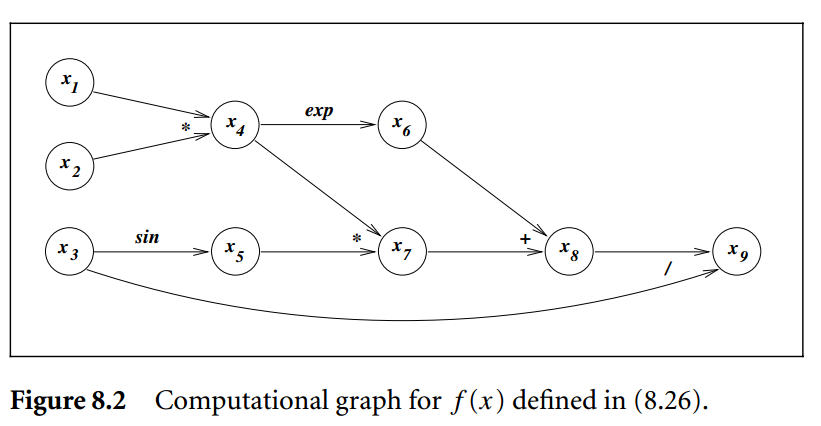
This works because we are looking at point values, not as a general expression. The dependency of *x8* on *x3* is taken care of later on, when the derivatives are multiplied together.

**Derivative Computation**

Each edge in the graph is a partial derivative. After all the partial derivatives have been computed, the gradient can be computed.

This computation multiplies all partial derivatives from the final node to the first node. If there is a split in the path, the partial derivatives of the two paths are added together.

For example:



Adding up these split paths is summarized in the following formula:

|  |  |
| --- | --- |
|  | (8.31) |

In the above example, the *f = x9*. The node *x8* is a child of *x6*.

**Discussion**

This formula is a result of the derivative addition and multiplication rules.

As a check:

Regardless of how *x11* and *x10* combines to form *f*, the formula is saying:

If , the calculus says:

which agrees with the rule since

If , the calculus says:

which agrees with the rule since

If , the calculus says:

which agrees with the rule since

**The algorithm**

During the functional evaluation phase, the values are evaluated in the direction of the arrow: {x1, (x11, x10), f}. During the point derivative evaluation phase, the values are evaluated in the reverse direction, {f, (x11, x10), x1} --- not all terms are always needed. f needs x10, which needs x1.

# System of Nonlinear Equations

**Nonlinear Equations:**

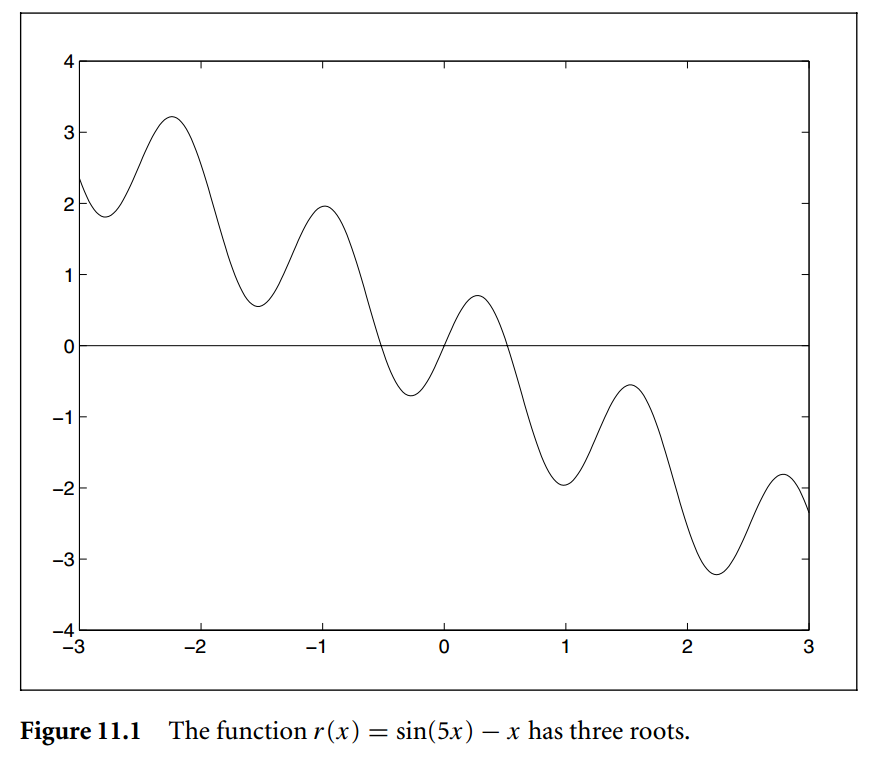
Example:

The has two inputs {*x1*, *x2*} and two outputs.

In general the is an Rn 🡪 Rn mapping.

**Root finding as an optimization problem**

Example:



You start at some point *x* and try to make *r(x)* zero. If *r(x)* is positive, then you do the steepest descent to try to reach the zero. Newton's methods for optimization can therefore be used for root finding as well.

Unless you start at near the root location, you are likely to get stuck in one of those peaks and valleys. Due to the large number of peaks and valleys, it's not likely to just randomly pick a point and converge to the true zero, using purely Newton's method.

**Solving a simpler problem first** --- in this case, one solution would be to solve first, then start the search for the zeros at x=0.

**Linear model for a system of equations:**

Suppose the system has two equations:

Near a particular point, the linear models would be:

In matrix form:

The Jacobian matrix J is a matrix of gradients.

**Newton's Method for Nonlinear Equations**

The linear model says:

For the current point , solve for :

|  |  |
| --- | --- |
|  | (11.6) |

If taking the full Newton step, the next point is then . That can oscillate, so to converge to the local minimum or maximum, take a partial step and check that the is getting closer to zero.

In the single variable case, when Newton's method arrive at a local minimum or maximum, it can no longer get closer to zero. The analogous case in the multivariable case is that the matrix becomes a non-invertible matrix.

## Broyden's Method

Secant methods, also known as quasi-Newton methods, calculates a fake Jacobian, called *B*, that satisfies the secant equation:

where

There are *n* conditions, but *n2* terms in the *B* matrix. Therefore, there are many update formulas possible.

Broyden's method:

|  |  |
| --- | --- |
|  | (11.28) |

The Broyden update makes the smallest possible change to the Jacobian as measured by .

**Proof**

Let *B* be any matrix that satisfies .

The final term in blue is one --- the 2-norm of the outer product is its inner product. Proof:

The is a scalar, so it can be factored out.

The maximum of the dot product is , when the angle between *x* and *s* is zero.

# Theory of Constrained Optimization

## Examples

At a **feasible point** *x*, the inequality constraint is **active** if *x* is on the border of the inequality.

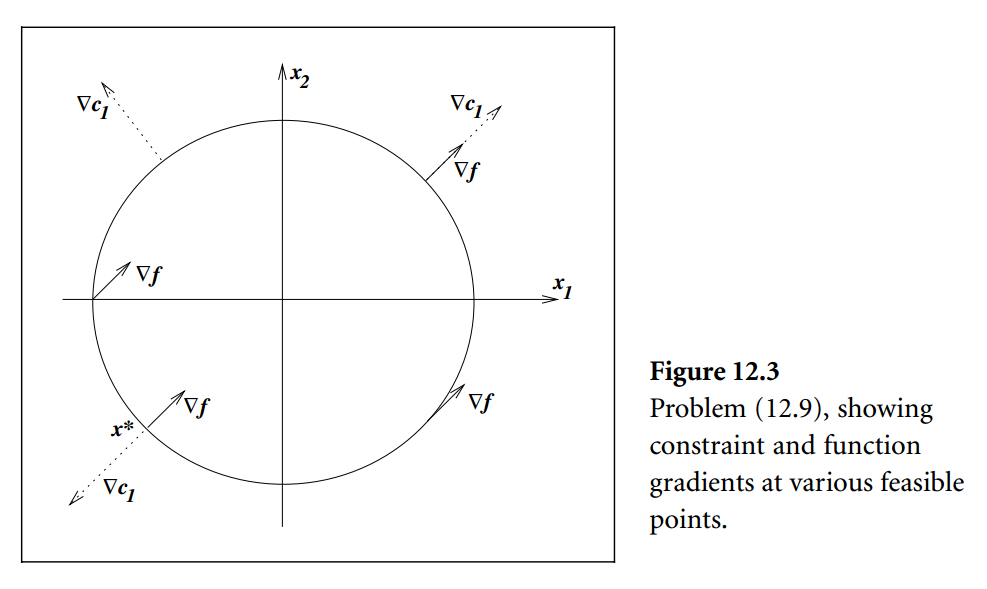
**Example 12.1 --- Two variable problem with a single equality constraint**

The to be minimized:

The constraint:

The feasible set is the circle of radius .

The minimum is at x\* = (-1, -1), at a point where is parallel to .



**Derivation of parallel to at the optimal point**

Suppose we are at point and intend to move in direction .

The equality constraint is . The linear model near point is:

To stay true to the constraint, we need and . This restricts to be:

The objective is to minimize . The linear model near point is:

To produce a decrease in we need . This restricts to be:

Putting these requirements together:

|  |  |  |
| --- | --- | --- |
| Requirement | Equation | Implication |
| Staying on |  |  |
| Decreasing |  | and forms obtuse angle |

As long as you can find an that is perpendicular to and >90 degrees from , you can decrease while staying on the constraint.

Geometrically speaking, if and are parallel, that is where you cannot find an . That is the optimal point.

**Lagrangian Function**

|  |  |
| --- | --- |
|  | (12.16) |

At the optimal point, ,

|  |  |
| --- | --- |
|  | (12.17) |

Note: In this situation, there is no restriction on the sign of *λ1*.

The condition (12.17) is necessary, but not sufficient. In example 12.1, both the maximum and minimum points satisfy (12.17).

**Example 12.2 --- A single inequality constraint**

The inequality is defining a circle region.

Case 1: inside the circle, the problem is an unconstrained optimization problem. We say the constraint is inactive.

Case 2: At the border of the circle, we say the constraint is activated.

The following reasoning is similar to before. We start with the constraint

We are at point and would like to move in the direction. The linear model is:

We are at the border, so

|  |  |  |
| --- | --- | --- |
| Requirement | Equation | Implication |
| Satisfying |  | or forms acute angle |
| Decreasing |  | and forms obtuse angle |

Unlike before, and being parallel, but pointing in opposite directions, is no longer optimal. In this case we can still find vector

and as long as we can find vector , that means the point is not optimal.

The only cease to exist if and are pointing in the same direction:

|  |  |
| --- | --- |
|  | (12.21) |

The Lagrangian function requirements:

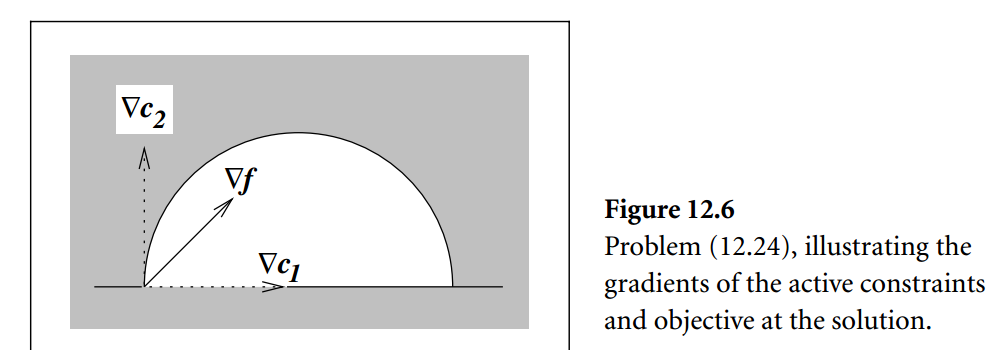
|  |  |
| --- | --- |
|  | (12.22) |
|  | (12.23) |

The equation (12.23) is to shutdown the Lagrangian function when we are inside the inequality region. When , the λ1 must be zero, and the Lagrangian function becomes just .

**Example 12.3 --- Two Inequality Constraints**

In this case we have two inequality constraints, .

The feasible set is a semicircle.



Several cases:

Inside the semicircle --- it's an unconstrained optimization problem

On the upper arc --- it's example 12.2, but with being the active constraint

On the lower line --- this time only is the active constraint.

The optimal is actually the corner . In this case, we have both constraints active.

We just add a new row to the requirement table:

|  |  |  |
| --- | --- | --- |
| Requirement | Equation | Implication |
| Satisfying |  | or forms acute angle |
| Satisfying |  | or forms acute angle |
| Decreasing |  | and forms obtuse angle |

A situation where this table cannot be satisfied is:

Dot product both sides with gives

The stated requirements and the table requirements will force . This breaks the last row of the table, and therefore there is no way to satisfy the whole table.

The coefficients show the relative impact of the constraints. So if λ2 >> λ1, then c2 is more important than c1, or that the optimal point is more sensitive to c2 than c1.

**In terms of the Lagrangian function:**

|  |  |
| --- | --- |
|  | (12.26) |

Note that the λ is a vector.

The inequality means all components of are .

If is not on the border of a constraint, then that constraint needs to be deactivated. This is done by:

So if we are on the arc of the semicircle, then the constraint is always positive. This requirement then force to be zero, and this deactivates the term in .

## First-Order Optimality Conditions

**KKT (Karush-Kuhn-Tucker) Conditions (Theorem 12.1)**

Given the optimization problem

|  |  |
| --- | --- |
| subject to | (12.1) |

where E (equality) and I (inequality) are two sets of indices.

and the Lagrangian function

|  |  |
| --- | --- |
|  | (12.33) |

the optimal conditions are:

|  |  |
| --- | --- |
|  | (12.34a) |
|  | (12.34b, c) |
|  | (12.34d) |
|  | (13.34e) |

Note that:

The requirement only applies to the inequality constraints.

The "turns off" the constraint when it's not active (when not on the boundary of the inequality).

## Duality

**The Primal Problem - it contains no equality constraints**

Material in this section is applicable to the problem

|  |  |
| --- | --- |
| subjected to | (12.81) |

Note there are no equality constraints. This enables all Lagrange multipliers .

and are convex.

The constraints are packed into vector form .

The Lagrange is written as .

**The Dual Problem**

The dual objective function:

|  |  |
| --- | --- |
|  | (12.82) |

The "inf" means the (greatest) lower bound. The "x" under the "inf" means varying "x" to get this lower bound.

So is the lower bound of the Lagrange function as a function of .

The dual problem to (12.81) is

|  |  |
| --- | --- |
| subject to | (12.84) |

**Example 12.10**

Consider the problem subject to .

For any particular value of λ, the lower bound is at

The lower bound function

The dual problem is

**is concave (Theorem 12.10)**

The function is concave and its domain *D* is convex.

Proof:

Recall that

For any and , and any α ϵ [0, 1]

Apply infimum to both sides. The infimum of a sum is greater than or equal to the sum of infimums.

The above is the definition of being concave.

The domain of is defined as the set such that .

The concave conclusion means that if the two ends and are both above -∞, then all points in between are above -∞.

**Weak Duality (Theorem 12.11)**

The optimal value of the dual problem (12.84) gives a lower bound on the optimal value for the primal problem (12.81).

For any feasible for (12.81), and any , we have .

Proof:

The is the infimum, so by definition .

The primal problem statement calls for , and having all constraints be inequality constraints mean , so the term .

Therefore .

**The primal problem and the dual problem has the same solution (Theorem 12.12) - solving the primal will solve the dual**

Suppose that is solution to the primal problem (12.81), and that satisfies the KKT conditions for the primal problem.

This same is also the solution to the dual problem (12.84)

Discussion:

The primal problem is solved by minimizing the Lagrange. Therefore the is actually the minimum point on the Lagrange function:

where is all possible vectors, while is by definition the solution to the primal problem.

The KKT condition says . When λi is non-zero, the corresponding constraint is active, so that particular .

Theorem 12.11 already said that , where is a feasible point, and is just anything .

In context of the current theorem, the and are linked by the KKT conditions and .

Out of all the , . This means is the maximum --- under the situation that the is being restricted by the KKT conditions.

**The dual problem and the primal problem has the same solution (Theorem 12.13) - solving the dual will solve the primal**

Suppose that solves the dual problem (12.84).

Suppose that the infimum in is attained at . The requirement for this stage is that is a convex function.

The is the solution of the primal problem (12.81).

Discussion:

This is the converse of theorem 12.12 --- solve the dual problem first, and then propagating from to .

Key requirement is that is a convex function, so that there is a one to one mapping between the in theorem 12.12 and the in theorem 12.13.

**Example 12.11 - Linear Programming Duality**

|  |
| --- |
| subject to |

If :

If :

requiring:

The dual problem is:

|  |
| --- |
| subject to |

**Example 12.12 - Convex Quadratic Programming Duality**

|  |
| --- |
| subject to |

where *G* is symmetric positive definite matrix.

The infimum is achieved when

Reminder: and

The term

with the requirement

The dual problem is:

|  |
| --- |
| subject to |

# The Simplex Method

**Standard Form of Linear Optimization Problems**

|  |  |
| --- | --- |
| , subject to | (13.1) |

**Transformation to the standard form**

Example:

, subject to

Add **slack variables** :

, subject to ,

, subject to ,

, subject to ,

Slack variables are also called **surplus variables**.

This is still not the standard form, since there is no . We are missing .

Split into positive and negative parts: , so we can require

The new constraint equations:

The new objective function:

Example: becomes

## Optimality

**Optimal Conditions**

Lagrangian function for , subject to

Regarding the constraints, when "unpacked" they looked like:

The portion of the Lagrange function having to do with this constraint looks like:

implies

|  |  |
| --- | --- |
|  | (13.4a) |

This uses the property

Doing this for the n=3 case:

Let

This multiplies out to be:

The full set of necessary conditions for x\* to be a solution of :

|  |  |
| --- | --- |
|  | (13.4a) |
|  | (13.4b, 13.4c, 13.4d) |
|  | (13.4e) |

Regarding :

When , is non-zero.

When , *si* is zero. This turns off the *si* term of the when not on the boundary.

because every pair is zero.

Although , the and vectors are individually non-zero.

## Geometry

**Basic feasible points**

The matrix *A* in the Simplex algorithm is wide --- *A* is matrix with . There are free variables.

The **basic feasible points** are solutions to the obtained in a very specific way.

Choose values in to be free variables and set them to zero. This is equivalent to setting columns of to zero.

Let the non-zero column indices be the set . The corresponding columns form the matrix *B*:

The *B* is a subset of *A*, and in general you can have many different *B* matrices, as in , , ... The *B* matrix is , and normally has a unique solution point.

Each from above, plus the zeroes from the deleted columns, corresponds to a full vector that is a solution to . These vectors are **basic feasible points**.

**solution as affine combinations of basic feasible points**

If are all solutions to , then

meaning

are also solutions. Note the weights have to add up to one. These are affine combinations.

**solution as convex combinations of basic feasible points**

For to be valid in a linear optimization problem, the entries need to be .

The different vectors come from different matrices, so they have different free variables set as zeros. For example

Let *A* be a matrix.

,

,

The affine combination

needs to have terms that are . This is forcing the weights and to be positive as well. If you have a negative weight, such as

But the last two terms of are stuck at zero, so there's no way can cancel out the -1 and -1.5.

So not only the weights have to add up to one, they have to be positive as well. This means the solution to in a linear optimization problem is a convex combination.

**solutions as a polytope**

The acceptable solutions to can be expressed as convex combination of the basic feasible points.

**Basic feasible points are vertices**

The vectors cannot be expressed as linear combination of other points in the solution set.

Suppose , , and some how .

The weights must be positive, so the form of and must be

If those first two terms are non-zero, there is no way to revert them back to zero in given the requirement that the weights are positive.

Therefore and must come from the same matrix as .

Geometrically vectors are endpoints in a line segment, and corners in a polygon.

**Relationship between hyperplane and polytope**

The system of equations can always be reduced to a single equation - so the solutions to , which is a polytope, can be reduced to a hyperplane.

Hyperplane world:

(1,2,3) is a solution

Polytope world:

(1,2,3, w=1) is a solution

There is a one to one correspondence between the two worlds.

The hyperplane is like a piece of paper, and the polytope is that piece of paper folded in complicated ways.

**Degenerate**

There are many assumptions in the discussion above. One is that the zeroes in the are unique, that only the free variables are zero. When the system is solved, there are no zeroes in the solution.

The indices of the non-zero columns, , is called the **basis**. The matrix B is called the **basis matrix**.

A basis is said to be **degenerate** if *xi* = 0 for some .

The linear optimization problem (13.1) is said to be degenerate if it has at least one degenerate basis.

## The Simplex Method

**Example 13.1**

subject to

- the indices where the *x* is non-zero  
*B* - the submatrix of A where the *x* is non-zero

- the indices where the *x* is zero  
*N* - the submatrix of A where the *x* is zero

A general theme of the simplex method is to break equations into *B* and *N* components.

The broken into *B* and *N* components:

Example 13.1:

From equation (13.4a), leads to

To break down into the B and N parts, you have to reorder the rows and columns as needed:

The order shown above mirrors the first iteration of Example 13.1.

Note you do not have two components for .

Let . This amounts to the non-degeneracy assumption, that the terms are non-zero.

|  |  |
| --- | --- |
|  | (13.19) |

Use the first equation to find the , and then the second equation to find .

Example 13.1:

**Zeroing another column (pivoting)**

Let the new iterate be and the current iterate be

The new iteration is bringing in *xq* - increasing this number from the current zero to some non-zero number.

The boxed part describes the new iteration.

- a vector made up of the non-zero entries from the old iteration, taking new values in the new iteration. So terms are all . As *x­q+* increases in the new iteration, the terms in will decrease, until eventually one of those terms will hit zero. So one of the terms will be zero.

|  |  |
| --- | --- |
|  | (13.22) |

**The effect of pivoting on the objective function**

The new value of the objective function is

substituting :

Using equation (13.19)

Also from equation (13.19), we have . The new index *q* is from set , so one term of this equation is:

The conclusion:

|  |  |
| --- | --- |
|  | (13.24) |

Where is the objective function from the old iteration, and is from the new iteration.

The objective function will decrease IF AND .

The simplest way to select *q* is to look at the and use its most negative term.

From equation (13.22)

Let . The vector describes how fast the terms will decline as increases and can predict the largest value that can be used.

Example 13.1

The largest value is -3. The indexing set , and -3 corresponds to index 1.

Applying equation (13.22):

So for every unit increase in , the will decrease by [1 2]T.

The farthest can go is to 4,

If goes beyond 4, the second term in the vector goes negative, and that violates the problem statement's requirement that .

One term of the vector went from 0 to 4, while another term went from 8 to 0. The total number of zeroes remain unchanged, and this new is a vertex just like the old .

**Unresolved Issues**

Linear algebra optimizations:

The algorithm solves two systems: . This is the same *B*, so LU factorization will help. Also, for each new iteration, *B* drops one column and replace that with a new column. If the terms in are re-ordered accordingly, the *B* is really changing just one column per iteration. It would be more efficient to update the LU rather than re-calculate it.

The choice of index *q*:

From equation (13.24) , the largest decrease in the objective function is not necessarily the most negative *sq*. It is actually the product of .

Finding the initial point is non-trivia.

Degenerate steps and cycling:

A degenerate step is when a term in is zero. The new step value is zero, because increasing it beyond zero will make the go below zero.

The problem is that both the old step and the new step has more zeroes than it's supposed to. Cycling happens when the algorithm is stuck taking one degenerate step after another - there's always that extra zero that forces the new to be zero. The objective function never decreases and the algorithm is failing to converge to an optimal solution.

Maybe the algorithm already found an optimal. Or maybe the previous choice of *q* is wrong and the algorithm needs to backtrack out of the infinite loop.

# Interior-Point Methods

## Primal-Dual Methods

**Problem Statement**

Linear programming problems in the standard form:

, subject to

The Lagrangian function:

The Karush-Kuhn-Tucker (KKT) conditions:

Newton's method is applied to the above equations. That last requirement, , is what makes this problem hard. Standard Newton's method will easily overstep the boundary.

Primal-dual methods are modified version of Newton's method and they generate iterates that satisfy the , note the lack of the equal sign. These intermediate points therefore stays in the interior of the feasible region, hence the name interior point methods.

The is a matrix function that summarizes the KKT conditions:

where .

**Standard Newton Step**

The setup is:

|  |  |
| --- | --- |
|  | (14.7) |

where:

The matrix on the left is the Jacobian of .

|  |  |
| --- | --- |
|  | (14.8) |

The vectors are residual vectors.

The step is .

This step is trying to reduce the *xisi* products straight to zero. The boundary information is not in the matrix.

**Modified Newton Step**

The modified Newton step tries to reduce the *xisi* products, on average, to a smaller number, and not straight to zero.

Let µ = average of the *xisi* products.

|  |  |
| --- | --- |
|  | (14.6) |

This makes µ a measure of desirability. As the algorithm iterates, the goal is to have µ approach zero.

The µ metric is called **duality measure**.

The modified Newton step:

|  |  |
| --- | --- |
|  | (14.9) |

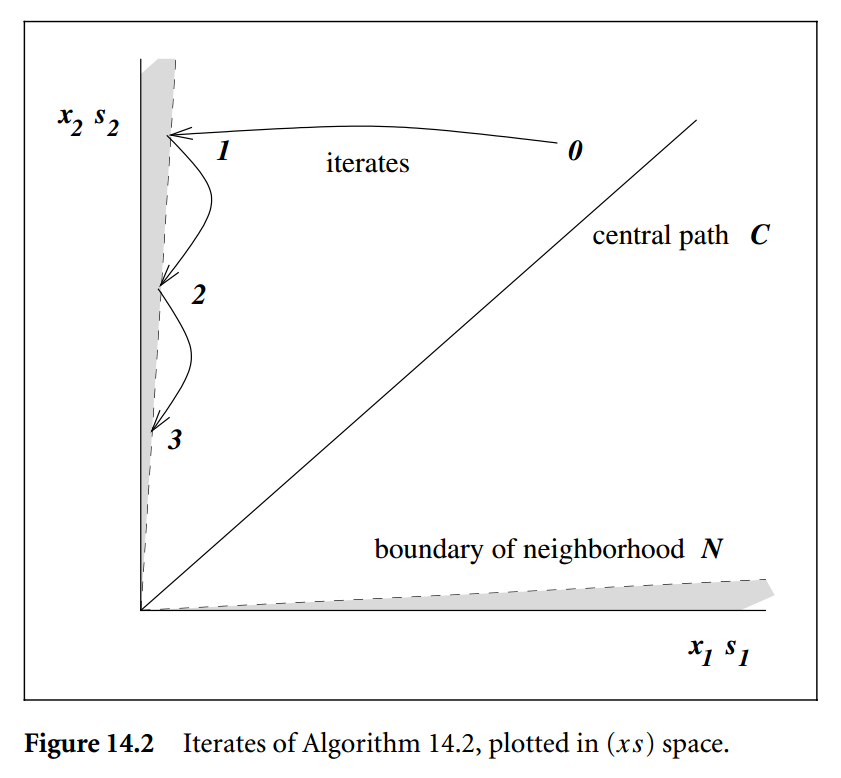
The act as a reduction factor.

So the current status is . The new step takes it to , rather than zero.

The step is .

The σ is called the **centering parameter**. The matrix produces *xisi* products. Using σ=1 aims for the same average for all these *xisi* products - IF the full Newton step is taken. Pictorially this tends to stay inside the interior of the feasible region.

On the other hand, using σ=0 reduces to the standard Newton step. This tends to head straight for the boundary region. The step leads to a larger µ reduction, but is less "centered".



**Barrier Function (Alternative Formulation)**

The original problem

, subject to

is restated as

, subject to

As , the . So using Newton's method on this formulation will also avoid the border. The is called a **logarithmic barrier**.

## Practical Primal-Dual Algorithms

**Nonlinear Correction for**

Among the KKT conditions is . This is a non-linear function, and using Newton's step will lead to an error.

The pure Newton setup is equation (14.7)

The Newton step is also called **affine-scaling**, which is why the steps have the "aff" superscript.

Specifically, the last row of the matrix equation is saying

On a term by term basis, we have

If we are current at , taking the full Newton step would lead to

instead of zero.

The corrected steps are

So the idea is to take the combined step instead of just the affine step.

In summary, first solve the pure Newton method:

Then solve

to get the real steps.

The left side of the two matrix equations are the same. So efficient factorization of the left side is important to the efficiency of this method.

**Centering parameter (σ) heuristic**

Calculate maximum allowable step lengths if affine-scaling direction is used:

Note there are two separate set of step lengths, one for and one for .

Define a metric that is the if the above two step lengths are used.

The centering parameter σ is adaptively chosen as

The *µ* is the current average. The is the degree of improvement. If *µ*aff is small compared with the current *µ*, then the affine step is making good progress, so there is less of a need for the centering parameter.

Conversely, if *µ*aff is large relative to the current *µ*, then standard Newton step is not working and the centering modification is needed.

**Step lengths**

The step lengths are chosen to be somewhere between 1 and the maximum possible step length.

where is chosen so that as we approach the solution.

# Nonlinear Constrained Optimization Fundamentals

**Problem Statement**

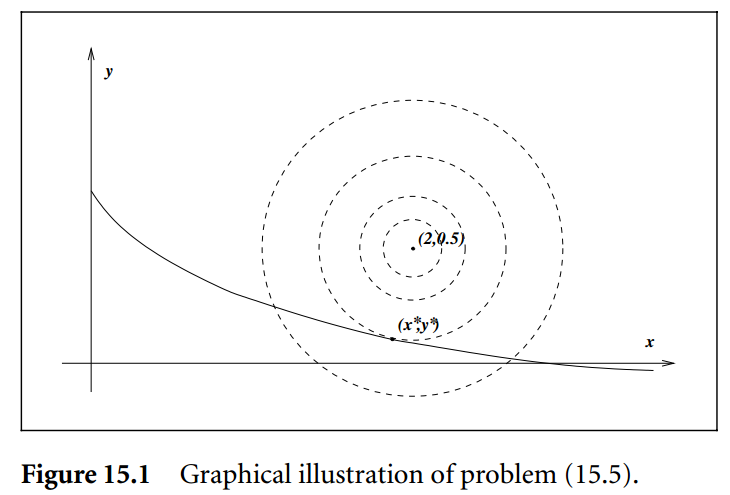
subject to

**Active-set Methods**

An active set method considers one subset of the constraints at a time. The current subset of constraints is called *W*, the **working set**.

Example 15.1

subject to



The global minimum of f(x,y) is at (2, 0.5). This is outside of the feasible zone though. The feasible points are between the (y=1/x) like curve, the x-axis, and the y-axis.

In looking for the optimal, go through the possible constraint combinations.

Let W = {}. This means all three constraints are off. The f(x,y) minimum is (2, 0.5), but that violates constraint #1 and is rejected.

Let W = {1, 2, 3}. This means all three constraints are on. But there is no point that lies on the intersection of all three constraints.

Let W = {1}. Solving the problem with just one constraint active will find the correct answer, which is labeled in the picture.

In this problem, each of the three constraints can be in an "on" or "off" condition. Meaning there are 23 = 8 possible combination of constraints to consider.

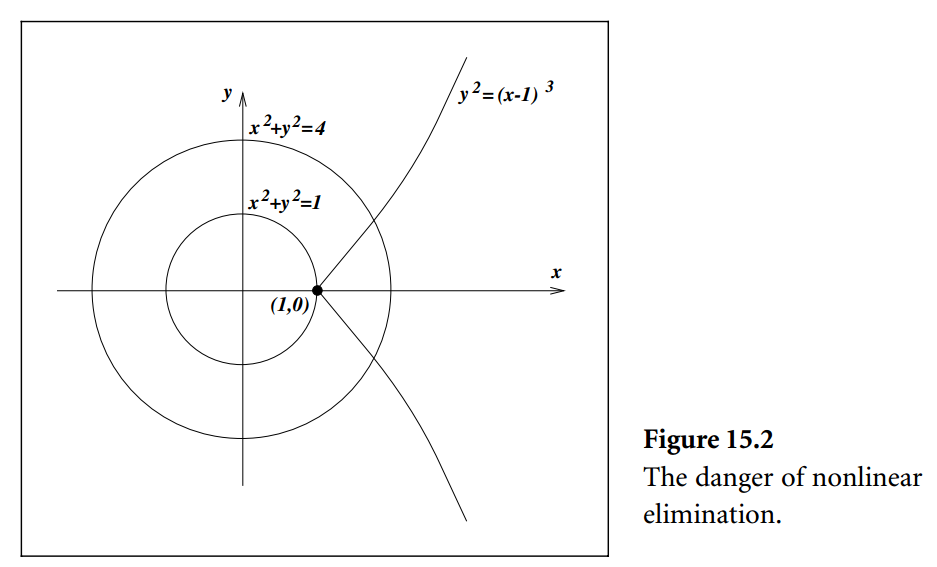
In general, the number of working set combinations is very high. The key in active set methods is to eliminate working sets.

**Elimination of variables - unsafe case**

The elimination of variables can subtly change the problem sometimes.

Example 15.2

subject to



The solution is (x,y) = (1,0).

The constraint is carrying an implicit restriction .

A direct substitution using leads to

The restriction disappears and the minimum becomes , when .

So in general, non-linear variable elimination is not safe.

**Variable elimination - linear case**

Variable elimination is safe for linear constraints.

Given the problem

subject to

where *A* is an matrix with .

There are more variables than there are equations, so some of those can be set as free variables.

We set the free variables. The remaining *m* variables can then be eliminated in .

Example 15.3

subject to

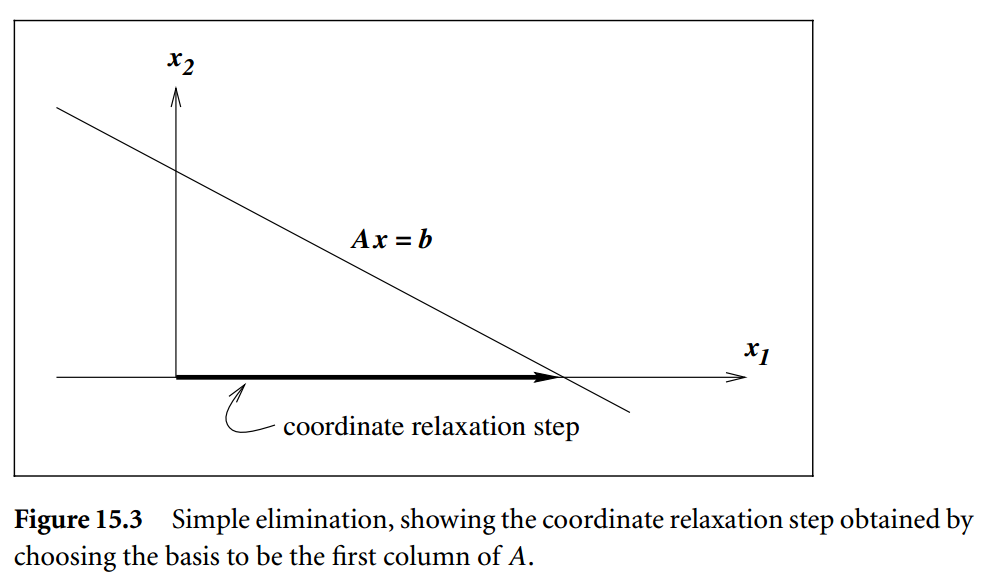
Choose as the variables of elimination since they are easy to solve for - let the other four be free variables.

The constraint is restated as

Divide the second row by 4

The objective function can then be reduced from to .

So you are choosing some set of variables and computing the rest . For a two-variable situation, this is a line



You choose *x2*, and then compute *x1*. This process is called **coordinate relaxation step**. The danger is that the line (or hyperplane) runs nearly parallel to the *x2*, resulting in a very large *x2*.

The challenge is in choosing the right sets of free variables, so that the remaining matrix is easy to solve.

Symbolically

where the columns are re-arranged so that the free variables are in and the elimination variables are in .

are called the **basic variables**, and the *B* matrix is called the **basis matrix** - the *B* matrix is being "used" to span Rm.

The challenge is to find the right set of columns, so that the does not become ill-conditioned.

**Elimination of variables - inequality constraints**

Elimination of variables does not always benefit inequality constraints.

A simple constraint might get turned into , making the situation even more complicated.

**Merit Functions**

A merit function is some combination of the objective function and the constraints .

A merit function would look like the Lagrangian function:

The KKT conditions call for

So maximizing (or minimizing) merit functions mirror this condition.

**Filters**

This approach finds a set of "good" points, instead of keeping a single best point.

The constrained optimization problem is broken down into two goals:

and

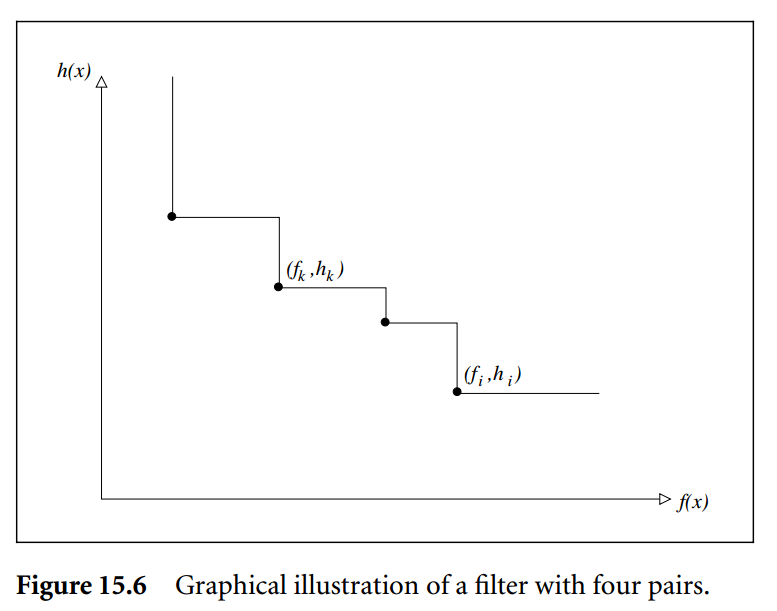
The is the objective function being minimized, while the is a measure of constraint violation. So the idea is to allow for some violation of the constraint, if it leads to a lower .

A pair **dominates** another pair if both and .

A **filter** is a list of pairs such that no pair dominates any other pair.

An iterate is **acceptable** to the filter if is not dominated by any pair in the filter.

So as the algorithm iterates, we keep track of a bunch of these pairs. For practical purposes, the and should differ by some meaningful amount, so it's like . This requirement is there so we don't have to keep track of a billion points. Graphically:



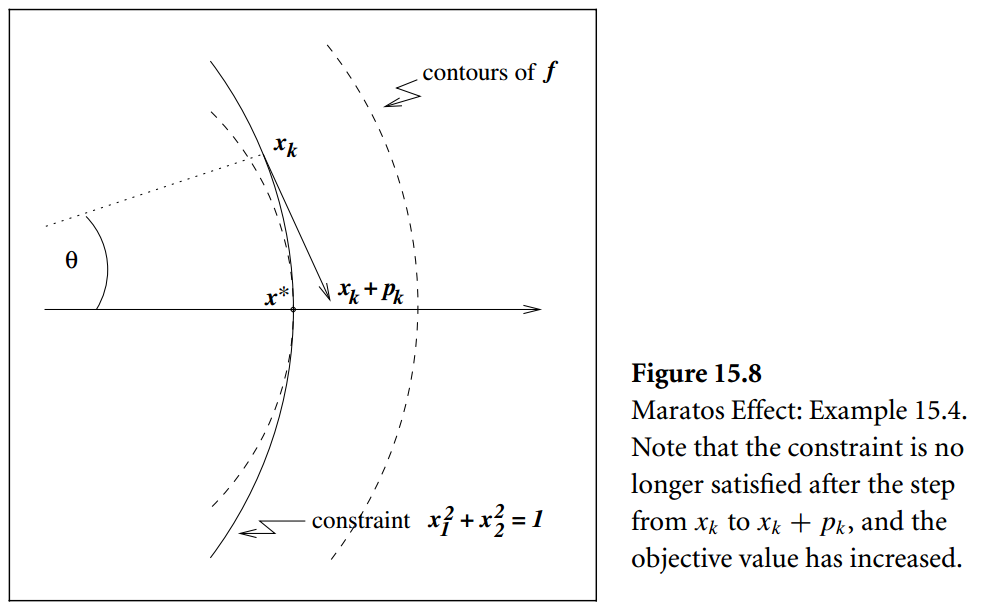
|  |  |
| --- | --- |
|  |  |
| 5 | 0 |
| -5 | 5 |

This notation is frequently used to express inequality constraint violation. The constraints in standard form are . So positive values mean no violation, and negative values mean violation.

**Maratos Effect**

This is a situation where a new iteration, , is mathematically closer to the final optimal point , but the objective function and constraint violation are both worse off.

Example 15.4



The constraint is the unit circle .

The two dashed lines show the contours of the objective function .

The current point, , is on the unit circle constraint. The is also quite close to in value - due to the contour of being so close by.

The new point, , is off the unit circle. It also is farther from the optimal contour. However, is geometrically closer to the final optimization point .

To avoid the Maratos effect:

1. Use a merit function that does not suffer from the Maratos effect.
2. Take an additional step, for a total of , to move back to the constraint (Example 15.4's unit circle).
3. **Nonmonotone strategy** - Allow the merit function to increase, but just for a few iterations.

# Quadratic Programming

**The general quadratic optimization problem**

subject to

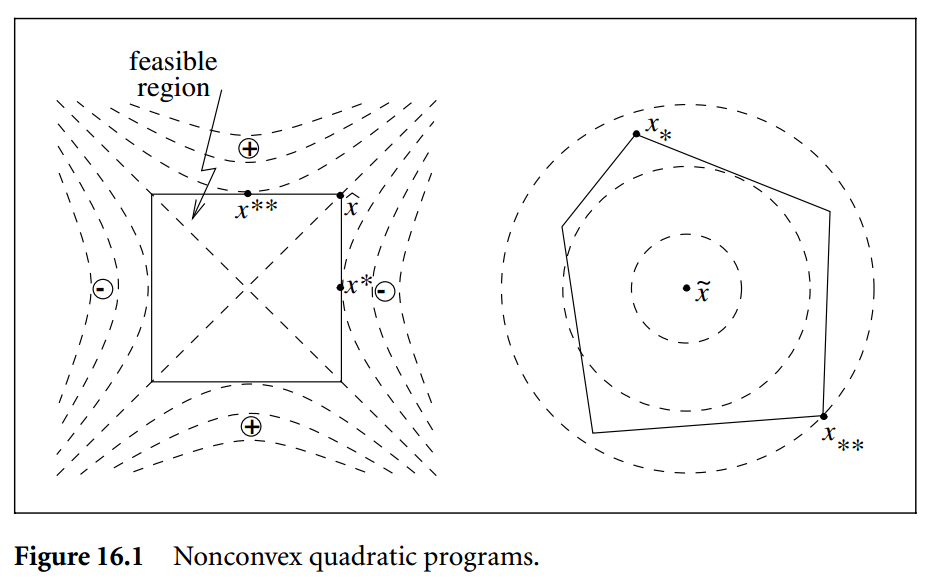
Note that *G* is a symmetric matrix.

**Convexity**

This chapter focus primarily on **convex quadratic programs** (optimization problems) --- this means *G* is positive semidefinite.

When *G* is not positive definite, the optimization problem is called "**nonconvex QP**" or "**indefinite QP**".

A **strict local minimizer**  means is smaller than all in the neighborhood. When *G* has a zero in its eigenvalues, the minimizer might not be strict.



In the picture on the left, the *G* has one positive and one negative eigenvalue. The + and - symbols indicate increase and decrease of the function in that general direction. The x\*\* is a local maximizer, x\* is a local minimizer, and the center of the box is a stationary point.

In the picture on the right, the *G* has negative eigenvalues. So the contour lines all show decrease. The center is a global maximizer, while and are local minimizers.

## Equality-Constrained Problems

**Problem statement**

subject to

*A* is assumed to have full row rank.

**Optimality conditions**

This is very similar to the material in the Simplex chapter, especially equation (13.4a).

What's new is just the .

implies

**Optimality conditions as a system of equations**

|  |  |
| --- | --- |
|  | (16.4) |

**Example 16.2**

Consider a quadratic optimization problem where

Setup:

Answer:

**Python for Example 16.2**

import numpy

import numpy.linalg as linalg

numpy.set\_printoptions(precision=3, # decimals to print out

suppress=True, # suppress scientific notation for small numbers

linewidth=100)

###########################################################

# Example 16.2

# From the problem statement

G = numpy.array([[6, 2, 1],

[2, 5, 2],

[1, 2, 4]])

c = numpy.array([[-8, -3, -3]]).T

A = numpy.array([[1, 0, 1],

[0, 1, 1]])

b = numpy.array([[3, 0]]).T

Z = numpy.zeros((A.shape[0], A.shape[0]))

# (K)(x) = (c\_b)

K = numpy.vstack((numpy.hstack((G, -A.T)),

numpy.hstack((A, Z))))

c\_b = numpy.vstack((-c, b))

linalg.solve(K, c\_b)

#array([[ 2.],

# [-1.],

# [ 1.],

# [ 3.],

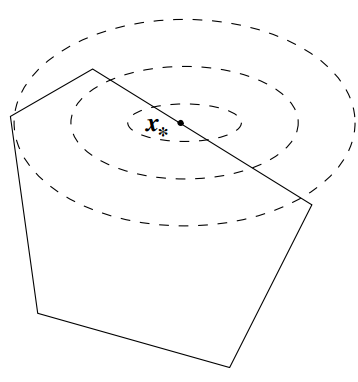
# [-2.]])

**Degeneracy in Inequality-Constrained Problems**

If inequality constraints are present, then they are only active if the solution x\* sits on the inequality boundary.

One degenerate situation is that the set of constraints is linearly dependent at the solution x\*.

Another situation is when the minimizer is right on one of the constraints.

We say that particular constraint is **weakly active**. It's active in the sense that is on the inequality boundary.

But is the minimum of the quadratic form, so the gradient for at point. In the equation, the λ for that constraint would be zero. There's no "contribution" from that constraint - so we cay it's weakly active.

## Active-Set Methods for Convex QPs

**Problem statement**

subject to

For each iteration, all of the equality constraints and some of the inequality constraints are imposed as equalities. This subset of active constraints is called the **working set** *W*.

**The subproblem of being at taking a step**

The new

The minimization problem

stated in terms of

The term and are the same due to *G* being symmetric, so .

Regroup the expression in terms of (fixed value) +

The first two terms is , the current level. The effect from the new step is itself a quadratic optimization problem.

|  |  |
| --- | --- |
| In conclusion, we are at and would like to take step such that  where  subject to , | (16.39) |

Adapt equation (16.4) to solve this problem

**Non-zero step ,**

If is a feasible point, then that is the next point.

If not, then we go as far as we can along the direction.

where *αk* is between 0 and 1.

The conditions included in the working set *Wk* is already satisfied, so need to check against the inequalities that are not listed in the working set.

For each requirement

we have to check

Assuming that is a feasible point, that means . If , then the inequality is always satisfied.

If , then there is an upper limit on

Do this check for each inequality constraint not in the working set. The final *αk* is the smallest of these upper limits.

The constraint that is responsible for the *αk* limit is called the **blocking constraint**. This constraint needs to be added to the working set *W* for the next iteration.

**, check λ values**

Eventually, the . An optimal point, relative to the current working set, has been found.

Now need to check to see if the current working set is indeed correct.

If the working set is correct, then the Lagrange multipliers for the active inequalities are all . This check is done using a subset version of equation (16.4)

The purpose is slightly different. When equation (16.4) is introduced, we are solving a bunch of equations for the optimality conditions and . Here, we think our is the optimal point, but we need to double check the values.

Using just the first row

Solve for . All the inequality constraints need to have . If this is indeed the case, then is indeed the optimal point .

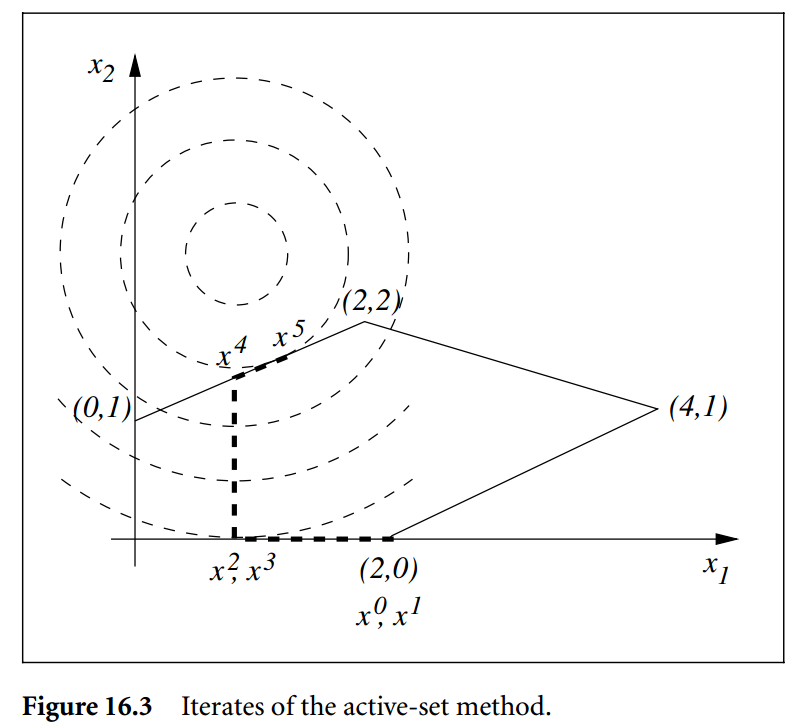
Reminder: there is no sign requirement for equality constraints.

If happens one or more times, drop the constraint associated with the most negative λ.

**Example 16.4**

The leads to quadratic form, but the problem statement has a factor in front.

subject to



|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | *W* |  |  |  |  |
| (2, 0) | {2, 4} | (0, 0) |  |  | (-2, -1) |
| (2, 0) | {4} | (-1, 0) | (1,1,1,1,1) | 1 |  |
| (1, 0) | {4} | (0, 0) |  |  | (-5) |
| (1, 0) | {} | (0, 2.5) | (0.6, 1,1,1,1) | 0.6 |  |
| (1, 1.5) | {0} | (0.4, 0.2) | (1,1,1,1,1) | 1 |  |
| (1.4, 1.7) | {0} | (0, 0) |  |  | (0.8) |

**Python**

###########################################################

# Example 16.4

G = numpy.array([[2, 0],

[0, 2.0]])

c = numpy.array([[-2, -5.0]]).T

A = numpy.array([[1, -1, -1, 1, 0],

[-2, -2, 2, 0, 1.0]]).T

b = numpy.array([[-2, -6, -2, 0, 0.0]]).T

tol = 1e-10

x\_k = numpy.array([[2, 0]]).T

# check for feasibility

feasible = True

r = A.dot(x\_k) - b

for i in range(0, len(r)):

if r[i] < 0-tol:

feasible = False

break

print("Point x\_k feasibility =", feasible)

# compute the working set W

W = []

r = A.dot(x\_k) - b

for i in range(0, len(r)):

if abs(r[i]) < tol: W.append(i)

# compute best direction p

g\_k = G.dot(x\_k) + c

A\_w = A[W]

# K = [[G -A^T], [A 0]] matrix

Z = numpy.zeros((A\_w.shape[0], A\_w.shape[0]))

K = numpy.vstack((numpy.hstack((G, -A\_w.T)),

numpy.hstack((A\_w, Z))))

# g\_k\_0 = [-g\_k, 0]

Z = numpy.zeros((A\_w.shape[0], 1))

g\_k\_0 = numpy.vstack((-g\_k, Z))

p\_lambda = linalg.solve(K, g\_k\_0)

p\_k = p\_lambda[:2]

# compute the best alpha (skip if p\_k == 0)

alpha\_list = []

for i in range(0, A.shape[0]):

if i in W: alpha\_list.append(1)

else:

a\_i = A[i]

if a\_i.T.dot(p\_k)[0] >= 0:

alpha\_list.append(1)

else:

b\_i = b[i]

alpha = (b\_i - a\_i.T.dot(x\_k))[0] / a\_i.T.dot(p\_k)[0]

if alpha > 1: alpha = 1

alpha\_list.append(alpha)

alpha = min(alpha\_list)

# working set update if alpha < 1

if alpha < 1:

# add blocking constraint that limits alpha

W.append(alpha\_list.index(alpha))

x\_k = x\_k + alpha \* p\_k

# don't do the following until p\_k == 0

# check lambda values

# (A^T)(lambda) = g\_k

if A[W].T.shape[1] >= 2:

# the following requires 2 variables to work

lambda\_vec = linalg.solve(A[W].T, g\_k)

else:

# there is just one lambda value

lambda\_val = g\_k[0][0] / A[W].T[0][0]

lambda\_vec = numpy.array([[lambda\_val]])

# as a check, (A[W].T)(lambda\_vec) = g\_k

diff = A[W].T.dot(lambda\_vec) - g\_k

if numpy.linalg.norm(diff) > tol:

raise Exception("Unable to calculate lambda vector")

# find smallest (most negative) lambda value

min\_index = 0

for i in range(1,len(lambda\_vec)):

if lambda\_vec[i] < lambda\_vec[min\_index]:

min\_index = i

# remove the most negative lambda from the working set

if lambda\_vec[min\_index] < 0:

del W[min\_index]

## Interior-Point Methods

There's a great deal of similarity with the interior-point method section on linear optimization problems.

**Problem statement**

subject to

where *G* is symmetric and positive semidefinite. This enables the KKT conditions to be sufficient.

Matrix *A* is .

Note the lack of equality constraints.

**KKT conditions**

The first equation is .

The second equation is the inequality constraint with a slack variable

The third equation deactivates constraints when the slack .

In the fourth equation, the due to all constraints being inequality constraints.

**KKT conditions in matrix form**

where *Y* = diag(y1, y2, ..., ym), Λ = diag(λ1, λ2, ..., λm), .

**Newton's method in matrix form**

where

***µ***

Like in the linear case, we define a metric *µ* that is the average value

**Mehrotra's predictor-corrector method**

First compute using basic Newton's method. Then compute:

|  |  |
| --- | --- |
|  | (16.67) |

The σ is determined using the same procedure as the linear case:

The step size during the affine stepping, αaff, is between 0 and 1. It's the largest alpha that maintain feasibility:

**Two separate step sizes can lead to increase in residual**

When interior point methods are applied to the linear optimization problem, two step sizes were used.

In the quadratic case, two step sizes might lead to increase in residual, causing potential divergence, and so only one step size is used.

The two step sizes scenario:

The definition is

The point is that is not always greater than . To see why, recall that is part of the modified Newton's equation:

The first row is

With this equation, it's possible to eliminate either or . The book eliminates the .

becomes

When a single step size is used, , and , so .

# Penalty and Augmented Lagrangian Methods

## Quadratic Penalty Method

**Problem statement**

subject to

The quadratic **penalty function** is defined as

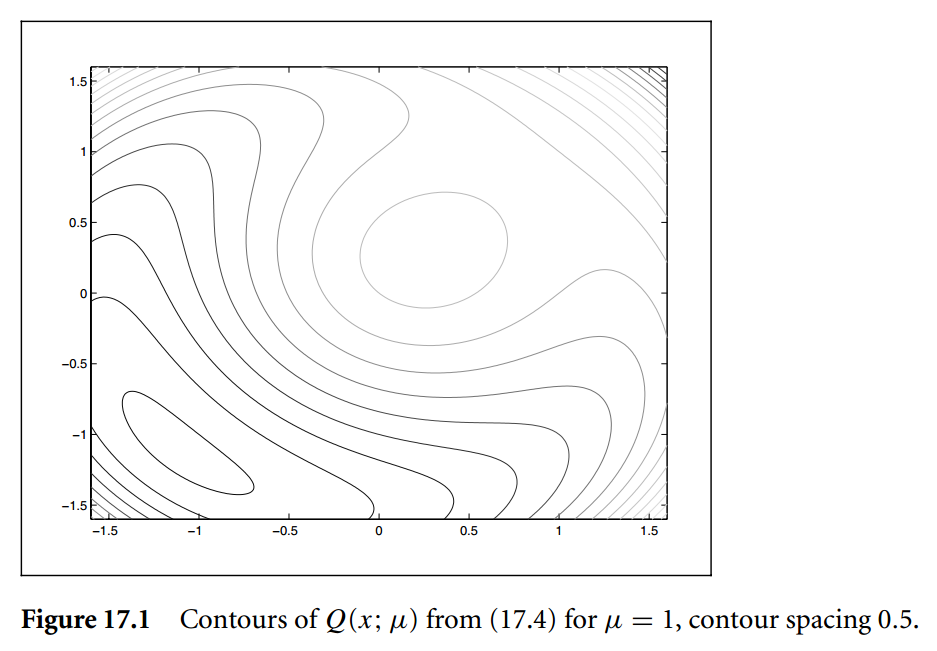
where µ > 0 is called the **penalty parameter**.

As µ is increased, the in should converge to the true minimum , but then the can become an ill-conditioned problem if µ is very large. Quasi-Newton methods mean subtractions, and subtracting large values, due to the large µ, is a problem.

**Example 17.1**

subject to

Contour plot for µ = 1:



There is a minimum at (-1.1, -1.1), near the true minimum of (-1, -1). Note that there is a local maximum near (0.3, 0.3).

**Motivation**

The approaches in the previous chapters assume linear constraints.

The approaches in this chapter combines the objective function with the constraints into a single merit function. This will work for non-linear constraints.

For problem with linear constraints, this chapter's approaches trade the constraint equations for a more complicated objective function.

**Unbounded penalty function - failure due to insufficient penalty**

The penalty function might be unbounded (below) even if the original problem has a minimum.

For example

subject to

has a minimum at (1, 0).

The penalty function is

If µ < 10, then the coefficient for x1 is negative, and has no lower bound.

This happens because the original constraint is an equality that locks down the value of x1 while the penalty function allows x1 to drift. If the penalty is not sufficient, then it's possible to drift very far away from the constraint. All penalty functions have same "insufficient penalty" failure potential.

**Inequality constraints**

The penalty function is . So any positive number is zero penalty, while a negative number like -5 counts as 5 penalty.

subject to

## Non-Smooth Penalty Functions

**The penalty function**

This penalty function is using the 1-norm of .

**Exactness of the penalty function**

The minimum for the quadratic penalty function is different from the minimum of the original problem - so we say the quadratic penalty function is not **exact**.

The penalty function is exact if , where . (Theorem 17.3)  
In practice you don't know what λ\* is, so you just try ever larger values of µ.

**Other norms**

Norms other than the 1-norm can be used.

The most common norms are 1-norm, 2-norm (not squared), and ∞-norm.

These penalty functions are exact for , where . The is the dual norm of .

## Augmented Lagrangian Method

**The augmented Lagrangian function**

So it's the Lagrangian function augmented by the quadratic penalty.

This function has better convergence for the quadratic penalty function --- meaning that the will approach the true minimum faster, for lower µ --- as long as the estimate λ is good. Intuitively, this is because the is closer to the KKT Lagrangian function "gold standard" than .

The book writes , but I feel the λ is a parameter that is updated per iteration, similar to µ.

**λ estimation**

With each iteration, the µ gets larger, and the λ is also re-estimated using

Discussion:

From the definition of :

From the definition of Lagrange function and the KKT conditions:

Matching the two equations

# Sequential Quadratic Programming

**Problem statement**

subject to

and

**KKT equations**

The Lagrangian function:

Gradient of the Lagrangian:

Let

KKT equations:

**Applying Newton's method to the KKT equations**

**Linearization of constraint**

Exercise 18.5

Given the constraint , the Newton's method linearizes the constraint via the equation

The derivative matrix is

So the gradient is expressed in row form

Let

The constraint , when linearized is

At (0, 1), the constraint is

Interior-Point Methods for Nonlinear Programming