In [7]:

```
using Plots
using Optim
using NLopt
```

```
ArgumentError: Package NLopt not found in current path:
- Run `import Pkg; Pkg.add("NLopt")` to install the NLopt package.


Stacktrace:
 [1] require(::Module, ::Symbol) at .\loading.jl:893
 [2] include_string(::Function, ::Module, ::String, ::String) at .\loading.jl:1091
```

Numerical Optimization

The general problem we consider is how to find the maximizing element $\hat{\theta}$ (a K -vector) of a function $s(\theta)$. This function may not be continuous, and it may not be differentiable. Even if it is twice continuously differentiable, it may not be globally concave, so local maxima, minima and saddlepoints may all exist. Supposing $s(\theta)$ were a quadratic function of θ, e.g.,

$$s(\theta) = a + b'\theta + \frac{1}{2}\theta' C\theta,$$

the first order conditions would be linear:

$$D_\theta s(\theta) = b + C\theta$$

so the maximizing (minimizing)\ element would be $\hat{\theta} = -C^{-1}b$. This is the sort of problem we have with linear models estimated by OLS. It's also the case for feasible GLS, since conditional on the estimate of the varcov matrix, we have a quadratic objective function in the remaining parameters.

More general problems will not have linear f.o.c., and we will not be able to solve for the maximizer analytically. This is when we need a numeric optimization method.

Search

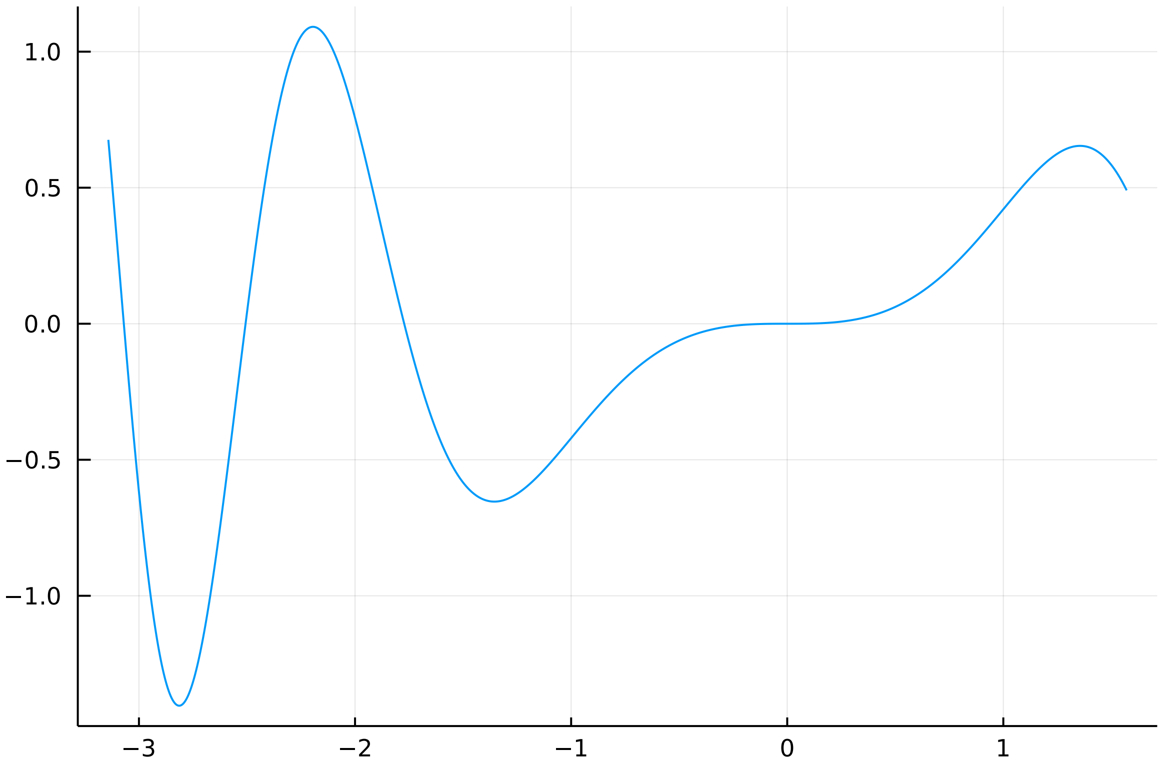
The idea is to create a grid over the parameter space and evaluate the function at each point on the grid. Select the best point. Then refine the grid in the neighborhood of the best point, and continue until the accuracy is "good enough". One has to be careful that the grid is fine enough in relationship to the irregularity of the function to ensure that sharp peaks are not missed entirely.

To check q values in each dimension of a K dimensional parameter space, we need to check q^K points. For example, if $q = 100$ and $K = 10$, there would be 100^{10} points to check. If 1000 points can be checked in a second, it would take 3.171×10^9 years to perform the calculations, which is approximately 2/3 the age of the earth. The search method is a very reasonable choice if K is small, but it quickly becomes infeasible if K is moderate or large.

In [16]:

```
# plot the line
    x = range(-pi,stop=pi/2.0,length=1000)
    obj = theta-> 0.5*theta*sin(theta^2.0)
    y = obj.(x)
    plot(x, y, legend=false)
```

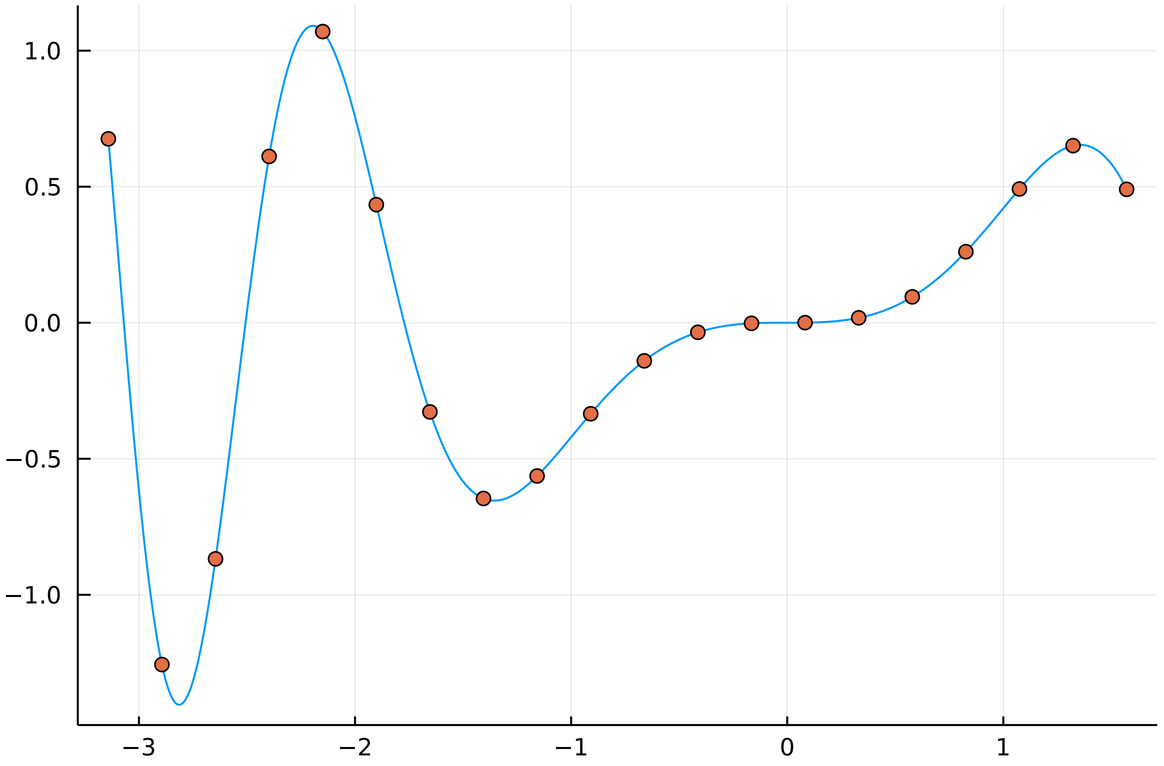
Out[16]:



In [17]:

```
points=20
    # plot the grid points
    x = range(-pi,stop=pi/2.0,length=points)
    y = obj.(x)
    scatter!(x, y)
```

Out[17]:



In [14]:

```
argmin(y)
```

Out[14]: 2

In [8]:

```
function GridExample(points)
    # plot the line
    x = range(-pi,stop=pi/2.0,length=1000)
    obj = theta-> 0.5*theta*sin(theta^2.0)
    y = obj.(x)
    plot(x, y, legend=false)
    # plot the grid points
    x = range(-pi,stop=pi/2.0,length=points)
    y = obj.(x)
    scatter!(x, y)
    # plot the best point found
    return scatter!([x[argmin(y)]], [y[argmin(y)]], markersize=10)
end
```

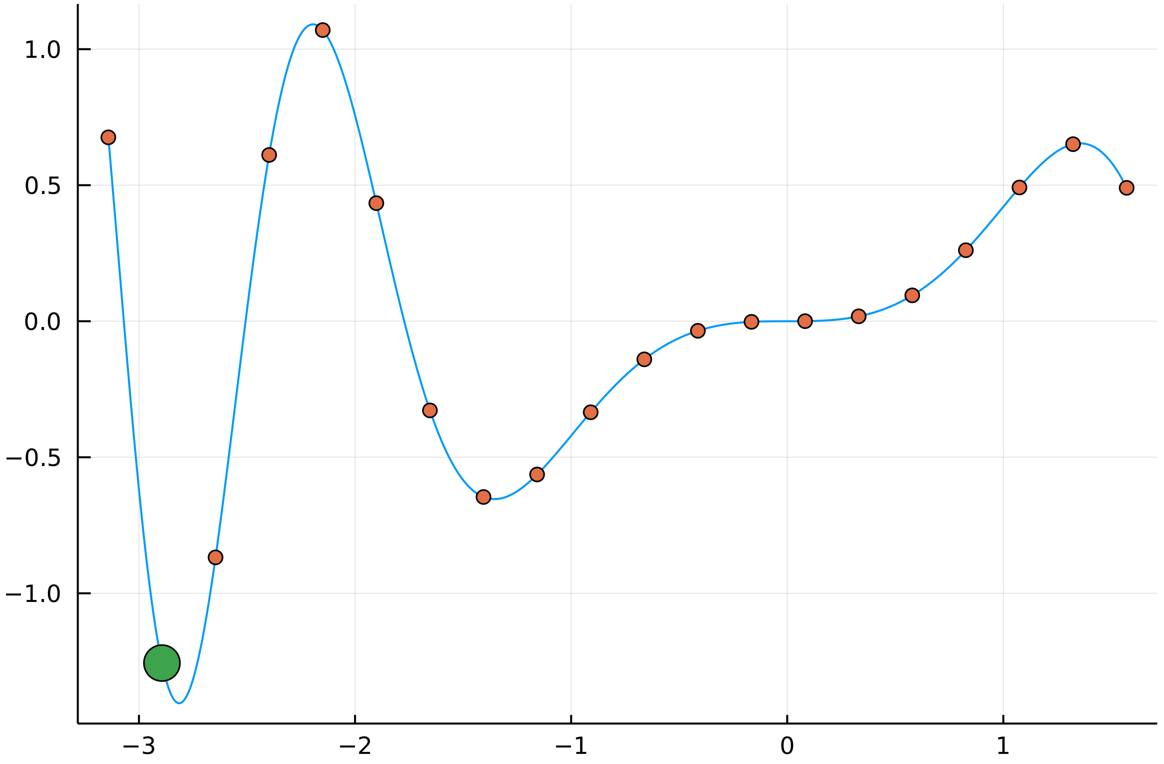
Out[8]:

```
GridExample (generic function with 1 method)
```

In [9]:

```
GridExample(20)
```

Out[9]:



Derivative-based methods

*In the following, the superscript k is used as the index of the iterations of a given method. It is not an exponent, and it is not the dimension of the parameter vector.

Reference: https://www.youtube.com/watch?v=N_ZRcLheNv0

We assume that the objective function is at least one time differentiable. Otherwise, these methods are not applicable, obviously. Derivative-based methods are defined by

1. the method for choosing the initial value, θ^1

2. the iteration method for choosing θ^{k+1} given that we're at θ^k at iteration k (based upon derivatives)
3. the stopping criterion.

The iteration method can be broken into two problems: choosing the stepsize a^k (a scalar) and choosing the direction of movement, d^k, which is of the same dimension of θ, so that

$$\theta^{(k+1)} = \theta^{(k)} + a^k d^k.$$

A locally increasing direction of search d is a direction such that

$$\frac{\partial s(\theta + ad)}{\partial a} > 0.$$

That is, if we go in direction d, we will improve on the objective function, at least if we don't go too far.

As long as the gradient at θ^k is not zero, there exist increasing directions, and they can all be represented as $Q^k g(\theta^k)$ where Q^k is a symmetric pd matrix and $g\left(\theta\right) = D_\theta s(\theta)$ is the gradient at θ. To see this, take a Taylor's series expansion around $a^0 = 0$

$$\begin{aligned} s(\theta + ad) &= s(\theta + 0d) + (a - 0)\, g(\theta + 0d)'d + o(1) \\ &= s(\theta) + ag(\theta)'d + o(1) \end{aligned}$$

For small enough a the $o(1)$ term can be ignored. If d is to be an increasing direction, we need $g(\theta)'d > 0$. Defining $d = Qg(\theta)$, where Q is positive definite, we guarantee that

$$g(\theta)'d = g(\theta)'Qg(\theta) > 0$$

unless $g(\theta) = 0$. Every increasing direction can be represented in this way (p.d. matrices are those such that the angle between g and $Qg(\theta)$ is less than 90 degrees).

With this, the iteration rule becomes

$$\theta^{(k+1)} = \theta^{(k)} + a^k Q^k g(\theta^k)$$

and we keep going until the gradient becomes zero, so that there is no increasing direction. The problem is now how to choose a and Q.

- Conditional on Q, choosing a is fairly straightforward. A simple line (1 dimensional grid) search is an attractive possibility, since a is a scalar. But there are other methods that may be better (bisection, golden, etc.).
- The remaining problem is how to choose Q.
- Note also that this gives no guarantees to find a global maximum.

Steepest Ascent

Reference: https://www.youtube.com/watch?v=TEB2z7ZlRAw

Steepest ascent (descent if we're minimizing) just sets Q to an identity matrix, since the gradient provides the direction of maximum rate of increase of the objective function.

- Advantages: fast - doesn't require anything more than first derivatives.
- Disadvantages: This doesn't always work too well however: see the Rosenbrock, or ''banana'' function

Newton's Method

Newton's method uses information about the slope and curvature of the objective function to determine which direction and how far to move from an initial point. Supposing we're trying to maximize $s_n(\theta)$. Take a second order Taylor's series approximation of $s_n(\theta)$ about θ^k (an initial guess).

$$s_n(\theta) \approx s_n(\theta^k) + g(\theta^k)' \left(\theta - \theta^k\right) + 1/2\left(\theta - \theta^k\right)' H(\theta^k) \left(\theta - \theta^k\right)$$

(g is the gradient vector and H is the Hessian matrix). To attempt to maximize $s_n(\theta)$, we can maximize the portion of the right-hand side that depends on θ, we can maximize

$$\tilde{s}(\theta) = g(\theta^k)'\theta + 1/2\left(\theta - \theta^k\right)' H(\theta^k) \left(\theta - \theta^k\right)$$

with respect to θ. This is a much easier problem, since it is a quadratic function in θ, so it has linear first order conditions. These are

$$D_\theta \tilde{s}(\theta) = g(\theta^k) + H(\theta^k) \left(\theta - \theta^k\right)$$

So the solution for the next round estimate is

$$\theta^{k+1} = \theta^k - H(\theta^k)^{-1} g(\theta^k)$$

However, it's good to include a stepsize, since the approximation to $s_n(\theta)$ may be bad far away from the maximizer $\hat{\theta}$, so the actual iteration formula is

$$\theta^{k+1} = \theta^k - a^k H(\theta^k)^{-1} g(\theta^k)$$

- A potential problem is that the Hessian may not be negative definite when we're far from the maximizing point. So $-H(\theta^k)^{-1}$ may not be positive definite, and $-H(\theta^k)^{-1}g(\theta^k)$ may not define an increasing direction of search. This can happen when the objective function has flat regions, in which case the Hessian matrix is very ill-conditioned (e.g., is nearly singular), or when we're in the vicinity of a local minimum, $H(\theta^k)$ is positive definite, and our direction is a decreasing direction of search. Matrix inverses by computers are subject to large errors when the matrix is ill-conditioned. Also, we certainly don't want to go in the direction of a minimum when we're maximizing. To solve this problem, Quasi-Newton methods simply add a positive definite component to $H(\theta)$ to ensure that the resulting matrix is positive definite, e.g., $Q = -H(\theta) + b\mathbf{I}$, where b is chosen large enough so that Q is well-conditioned and positive definite. This has the benefit that improvement in the objective function is guaranteed.
- Another variation of quasi-Newton methods is to approximate the Hessian by using successive gradient evaluations. This avoids actual calculation of the Hessian, which is an order of magnitude (in the dimension of the parameter vector) more costly than calculation of the gradient. They can be done to ensure that the approximation is p.d. DFP and BFGS are two well-known examples

```
In [30]: rosenbrock(x) =  (1.0 - x[1])^2 + 100.0 * (x[2] - x[1]^2)^2
         result = optimize(rosenbrock, zeros(2), BFGS())
```

```
Out[30]:  * Status: success

          * Candidate solution
             Final objective value:     5.471433e-17

          * Found with
             Algorithm:     BFGS

          * Convergence measures
             |x - x'|               = 3.47e-07 ≤/0.0e+00
             |x - x'|/|x'|          = 3.47e-07 ≤/0.0e+00
             |f(x) - f(x')|         = 6.59e-14 ≤/0.0e+00
             |f(x) - f(x')|/|f(x')| = 1.20e+03 ≤/0.0e+00
             |g(x)|                 = 2.33e-09 ≤ 1.0e-08

          * Work counters
             Seconds run:   0  (vs limit Inf)
             Iterations:    16
             f(x) calls:    53
             ∇f(x) calls:   53
```

Stopping Criteria

The last thing we need is to decide when to stop. A digital computer is subject to limited machine precision and round-off errors. For these reasons, it is unreasonable to hope that a program can **exactly** find the point that maximizes a function. We need to define acceptable tolerances. Some stopping criteria are:

- Negligible change in parameters: $4|\theta_j^k - \theta_j^{k-1}| < \varepsilon_1, \forall j4$
- Negligible relative change:

$$|\frac{\theta_j^k - \theta_j^{k-1}}{\theta_j^{k-1}}| < \varepsilon_2, \forall j$$

- Negligible change of function:

$$|s(\theta^k) - s(\theta^{k-1})| < \varepsilon_3$$

- Gradient negligibly different from zero:

$$|g_j(\theta^k)| < \varepsilon_4, \forall j$$

- Or, even better, check all of these.
- Also, if we're maximizing, it's good to check that the last round (real, not approximate) Hessian is negative definite.

Starting Values

The Newton-Raphson and related algorithms work well if the objective function is concave (when maximizing), but not so well if there are convex regions and local minima or multiple local maxima.

The algorithm may converge to a local minimum or to a local maximum that is not optimal. The algorithm may also have difficulties converging at all.

- The usual way to ENSURE that a global maximum has been found is to use many different starting values, and choose the solution that returns the highest objective function value. **THIS IS IMPORTANT IN PRACTICE**
- an alternative is to use a global optimization algorithm, e.g., simulated annealing or others, which may or may not be gradient based.

Calculating Derivatives

The Newton-Raphson algorithm requires first and second derivatives. It is often difficult to calculate derivatives (especially the Hessian) analytically if the function $s_n(\cdot)$ is complicated. Possible solutions are to calculate derivatives numerically, or to use programs such as MuPAD or Mathematica to calculate analytic derivatives.

Numeric derivatives are less accurate than analytic derivatives, and are usually more costly to evaluate. Both factors usually cause optimization programs to be less successful when numeric derivatives are used.

- One advantage of numeric derivatives is that you don't have to worry about having made an error in calculating the analytic derivative. When programming analytic derivatives it's a good idea to check that they are correct by using numeric derivatives. This is a lesson I learned the hard way when writing my thesis.
- there are also methods for automatic differentiation. This will probably become important in the future, but existing econometric software makes little use of it.

Numeric second derivatives are much more accurate if the data are scaled so that the elements of the gradient are of the same order of magnitude. Example: if the model is $y_t = h(\alpha x_t + \beta z_t) + \varepsilon_t$, and estimation is by NLS. Let $g()$ be the derivative of $h()$.

- $s_n(\theta) = \frac{1}{n}\sum_t \left(y_t - h(\alpha x_t + \beta z_t)\right)^2$ and
- $D_\alpha s_n(\cdot) = \frac{1}{n}\sum_t 2\left(y_t - h(\alpha x_t + \beta z_t)\right) g(\alpha x_t + \beta z_t) x_t.$
- $D_\beta s_n(\cdot) = \frac{1}{n}\sum_t 2\left(y_t - h(\alpha x_t + \beta z_t)\right) g(\alpha x_t + \beta z_t) z_t$
- suppose that $D_\alpha s_n(\cdot) = 1000$ and $D_\beta s_n(\cdot) = 0.001.$ One could define $\alpha^* = 1000\alpha;$ $x_t^* = x_t/1000;\beta^* = \beta/1000;\, z_t^* = 1000 z_t.$
- then $D_{\alpha^*} s_n(\cdot) = \frac{1}{n}\sum_t 2\left(y_t - h(\alpha^* x_t^* + \beta^* z_t^*)\right) g(\alpha^* x_t^* + \beta^* z_t^*) x_t^*.$ Everything is the same as before, except there is an x_t^* at the end, which causes the derivative to be 1 now.
- the same is true for the other derivative, it will be 1.
- this scaling causes the derivatives to be of the same order.

In general, estimation programs always work better if data is scaled in this way, since roundoff errors are less likely to become important. THIS IS IMPORTANT IN PRACTICE. In the future, if you start to do empirical work and get results that seem meaningless or crazy, try to remember this point.

- There are algorithms (such as BFGS and DFP) that use the sequential gradient evaluations to build up an approximation to the Hessian. The iterations are faster because the actual Hessian isn't calculated, but more iterations usually are required for convergence. Versions of BFGS are probably the most widely used optimizers in econometrics.

- Switching between algorithms during iterations is sometimes useful.

In [32]:

```
"""
    xopt, fopt, converged = fminunc(obj, startval)

Minimize the function obj, starting at startval.

fminunc() with no arguments will run an example, execute edit(fminunc,()) to see the co
fminunc() uses NLopt.jl  to do the actual minimization.

"""
function fminunc(obj, x; tol = 1e-08)
    results = Optim.optimize(obj, x, LBFGS(),
                            Optim.Options(
                            g_tol = tol,
                            x_tol=tol,
                            f_tol=tol))
    return results.minimizer, results.minimum, Optim.converged(results)
    #xopt, objvalue, flag = fmincon(obj, x, tol=tol)
    #return xopt, objvalue, flag
end

function fminunc()
    println("with no arguments, fminunc() runs a simple example")
    println("type edit(fminunc, ()) to see the example code")
    # return of objective should be real valued, thus the [1] to pull value out of 1-di
    obj = x -> (x.^2)[1]
    # obj = x -> x'x # this would also work, as returns a Float64
    x = [2.0] # argument to objective function should be a vector, thus the brackets
    results = fminunc(obj, x)
end
```

Out[32]:

```
fminunc (generic function with 2 methods)
```

In [34]:

```
fminunc()
```

```
with no arguments, fminunc() runs a simple example
type edit(fminunc, ()) to see the example code
```

Out[34]:

```
([6.622924431098909e-12], 4.3863128020046805e-23, true)
```

In [38]:

```
"""
    xopt, fopt, converged = fmincon(obj, startval)

Minimize the function obj, starting at startval.

fminunc() with no arguments will run an example, execute edit(fminunc,()) to see the co
fminunc() uses NLopt.jl to do the actual minimization.

"""
function fmincon(obj, startval, R=[], r=[], lb=[], ub=[]; tol = 1e-10, iterlim=0)
    # the objective is an anonymous function
    function objective_function(x::Vector{Float64}, grad::Vector{Float64})
        obj_func_value = obj(x)[1,1]
        return(obj_func_value)
    end
    # impose the linear restrictions
    function constraint_function(x::Vector, grad::Vector, R, r)
```

```
        result = R*x .- r
        return result[1,1]
    end
    opt = Opt(:LN_COBYLA, size(startval,1))
    min_objective!(opt, objective_function)
    # impose lower and/or upper bounds
    if lb != [] lower_bounds!(opt, lb) end
    if ub != [] upper_bounds!(opt, ub) end
    # impose linear restrictions, by looping over the rows
    if R != []
        for i = 1:size(R,1)
            equality_constraint!(opt, (theta, g) -> constraint_function(theta, g, R[i:i
        end
    end
    xtol_rel!(opt, tol)
    ftol_rel!(opt, tol)
    maxeval!(opt, iterlim)
    (objvalue, xopt, flag) = NLopt.optimize(opt, startval)
    return xopt, objvalue, flag
end

function fmincon()
    println("with no arguments, fmincon() runs a simple example")
    println("type edit(fmincon, ()) to see the example code")
    # return of objective should be real valued, thus the [1] to pull value out of 1-di
    obj = x -> x'x
    x = [2.0, 2.0]
    # sum of params should be 1
    R = [1.0 1.0]
    r = 1.0
    results = fmincon(obj, x, R, r)
end
```

Out[38]:

```
fmincon (generic function with 6 methods)
```

In [39]:

```
fmincon()
```

```
with no arguments, fmincon() runs a simple example
type edit(fmincon, ()) to see the example code
```

Out[39]:

```
([0.49999750305193874, 0.5000024969480613], 0.5000000000124696, :FTOL_REACHED)
```

Example (Restricted Nerlove)