Dense-and-Sparse

September 7, 2017

```
In [1]: using PyPlot, Interact
```

1 Large-scale linear algebra: Dense matrix methods

The basic problem with most of the linear algebra techniques we have learned so far is that they **scale badly for large matrices**. Ordinary Gaussian elimination (LU factorization), Gram–Schmidt and other QR factorization algorithms, and techniques that computes *all* the eigenvalues and eigenvectors, all require $\sim n^3$ operations and $\sim n^2$ storage for $n \times n$ matrices.

This all assumes that you explicitly store and compute with all of the entries of the matrix, regardless of their values. Hence, they are sometimes called **dense matrix** algorithms (the opposite of "sparse" matrices, discussed below).

So, doubling the size of the matrix *asymptotically* requires about 8× more time. For any *finite* n, it is not quite a factor of 8 because *computers are complicated*; e.g. for larger matrices, it can use multiple processors more efficiently:

```
In [2]: A1 = randn(500,500)
        A2 = randn(1000,1000)
        @elapsed(lufact(A2)) / @elapsed(lufact(A1))

Out[2]: 7.27003410105763

In [3]: @elapsed(qrfact(A2)) / @elapsed(qrfact(A1))

Out[3]: 12.387061180927192

In [4]: @elapsed(eigfact(A2)) / @elapsed(eigfact(A1))

Out[4]: 5.708597794682041

In [5]: @time lufact(A2)
        @time eigfact(A2);

0.016880 seconds (11 allocations: 7.637 MB)
  2.827680 seconds (85 allocations: 30.831 MB, 2.65% gc time)
```

Still, if we take the $O(n^3)$ scaling as a rough guide, this would suggest that LU-factorizing (`lufact`) a $10^6 \times 10^6$ matrix would take $0.02\text{sec} \times 1000^3 \sim$ months and finding the eigenvectors and eigenvalues (`eigfact`) would take $2.6\text{sec} \times 1000^3 \sim$ decades.

In practice, we actually usually **run out of space before we run out of time**. If we have 16GB of memory, the biggest matrix we can *store* (each number requires 8 bytes) is $8n^2$ bytes $= 16 \times 10^9 \implies 40000 \times 40000$.

2 Sparse Matrices

The saving grace is that most *really* large matrices are **sparse = mostly zeros** (or have some other special structure with similar consequences). You only have to **store the nonzero entries**, and you can **multiply matrix × vector quickly** (you can skip the zeros).

In Julia, there are many functions to work with sparse matrices by only storing the nonzero elements. The simplest one is the `sparse` function. Given a matrix A, the `sparse(A)` function creates a special data structure that only stores the nonzero elements:

```
In [6]: A = [  2 -1  0  0  0  0
              -1  2 -1  0  0  0
               0 -1  2 -1  0  0
               0  0 -1  2 -1  0
               0  0  0 -1  2 -1
               0  0  0  0 -1  2]
```

```
Out[6]: 6×6 Array{Int64,2}:
          2  -1   0   0   0   0
         -1   2  -1   0   0   0
          0  -1   2  -1   0   0
          0   0  -1   2  -1   0
          0   0   0  -1   2  -1
          0   0   0   0  -1   2
```

```
In [7]: sparse(A)
```

```
Out[7]: 6×6 sparse matrix with 16 Int64 nonzero entries:
                [1, 1]  =  2
                [2, 1]  =  -1
                [1, 2]  =  -1
                [2, 2]  =  2
                [3, 2]  =  -1
                [2, 3]  =  -1
                [3, 3]  =  2
                [4, 3]  =  -1
                [3, 4]  =  -1
                [4, 4]  =  2
                [5, 4]  =  -1
                [4, 5]  =  -1
                [5, 5]  =  2
                [6, 5]  =  -1
                [5, 6]  =  -1
                [6, 6]  =  2
```

(Of course, in practice you would want to create the sparse matrix directly, rather than first making the "dense" matrix `A` and then converting it to a sparse data structure.)

We've actually seen this several times in **graph/network-based problems**, where we often get matrices of the form:

$$A = G^T DG$$

where D is diagonal (very sparse!) and G is the incidence matrix. Since each graph node is typically only connected to a few other nodes, **G is sparse** and so is A.

If each node is connected to a bounded number of other nodes (say, ≤ 20), then A only has $\sim n$ (i.e. *proportional to n, not equal to n*) entries, and Ax can be computed in $\sim n$ operations and $\sim n$ storage (unlike $\sim n^2$ for a general matrix).

So, a $10^6 \times 10^6$ sparse matrix might be stored in only a few megabytes and take only a few milliseconds to multiply by a vector.

Much of large-scale linear algebra is about devising techniques to exploit sparsity or **any case where matrix × vector is faster than $\mathbf{n^2}$**.

3 Scalable computation of eigenvalues

In fact, we already learned one algorithm that works well for sparse matrices: the power method to compute eigenvalues and eigenvectors. If we just repeatedly multiply a random vector by A, it converges towards the eigenvector of the largest $|\lambda|$. And since this only involves matrix × vector operations, it can take advantage of sparse matrices.

Moreover, there are variants of this algorithm that work for the smallest eigenvalues as well, and it turns out that there are more sophisticated variants that converge even faster than power iterations. In Julia, these are provided by the `eigs` function, which lets you compute a **few** of the biggest or smallest eigenvalues quickly even for huge sparse matrices.

3.1 Example

As an example, let's consider the **two-dimensional grid of masses and springs** that I showed in an earlier lecture, whose eigenvectors are the **vibrating modes**.

This can be thought of as a discretized approximation of a **vibrating drum**, which is described by the partial differential equation $\nabla^2 h = \frac{\partial^2 h}{\partial t^2}$ where $h(x,y,t)$ is the height of the drum surface (= zero at the edges of the drum). (This is an example taken from the class 18.303: Linear Partial Differential Equations at MIT.)

For 18.06, don't worry too much about how the matrix is constructed.

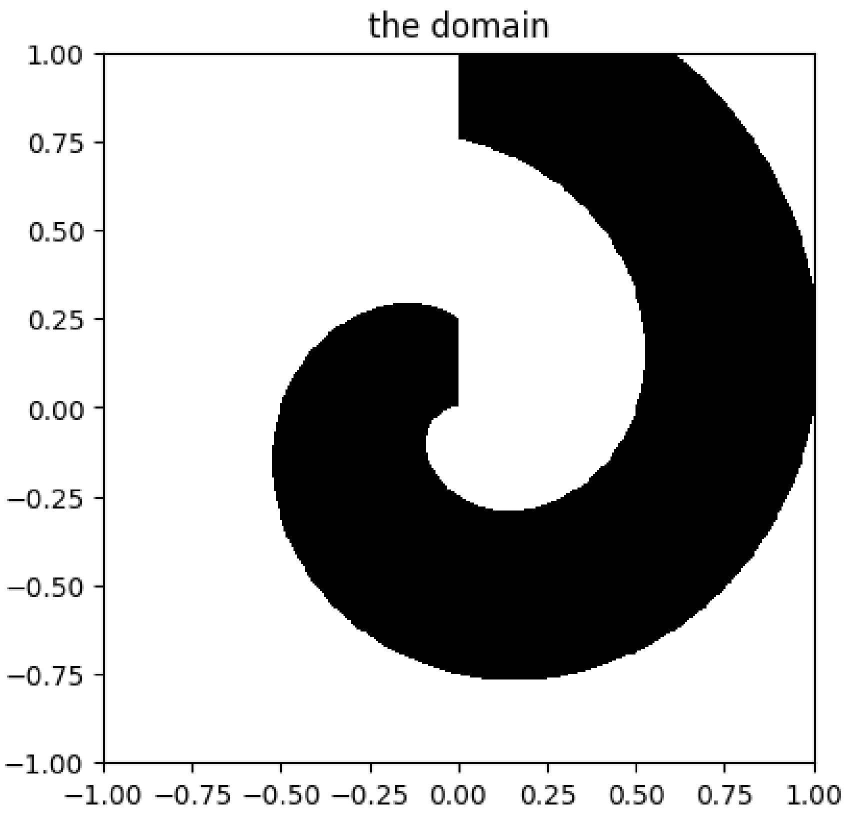
```
In [8]: # compute the first-derivative finite-difference matrix
        # for Dirichlet boundaries, given a grid x[:] of x points
        # (including the endpoints where the function = 0!).
        function sdiff1(x)
            N = length(x) - 2
            dx1 = Float64[1/(x[i+1] - x[i]) for i = 1:N]
            dx2 = Float64[-1/(x[i+1] - x[i]) for i = 2:N+1]
            spdiagm((dx1,dx2), (0,-1), N+1, N)
        end

        flatten(X) = reshape(X, length(X))

        # compute the -∇⋅ c ∇ operator for a function c(x,y)
        # and arrays x[:] and y[:] of the x and y points,
        # including the endpoints where functions are zero
        # (i.e. Dirichlet boundary conditions).
        function Laplacian(x, y, c = (x,y) -> 1.0)
            Dx = sdiff1(x)
            Nx = size(Dx,2)
            Dy = sdiff1(y)
            Ny = size(Dy,2)

            # discrete Gradient operator:
            G = [kron(speye(Ny), Dx); kron(Dy, speye(Nx))]

            # grids for derivatives in x and y directions
            x′ = [0.5*(x[i]+x[i+1]) for i = 1:length(x)-1]
            y′ = [0.5*(y[i]+y[i+1]) for i = 1:length(y)-1]
```

```
    # evaluate c(x)
    C = spdiagm([ flatten(Float64[c(X,Y) for X in x′, Y in y[2:end-1]]);
                  flatten(Float64[c(X,Y) for X in x[2:end-1], Y in y′]) ])

    return G' * C * G # -∇⋅ c ∇
end
```

```
Out[8]: Laplacian (generic function with 2 methods)
```

The above code defines the matrix for a box-shaped drum, but for fun I will change it to define a drum over an oddly shaped domain:

```
In [9]: N = 400
        x = linspace(-1,1,N+2)[2:end-1]
        y = x'  # a row vector
        r = sqrt(x.^2 .+ y.^2)      # use broadcasting (.+) to make Nx x Ny matrix of radii
        θ = broadcast(atan2, y, x) # and angles
        φ = exp(-(r - θ*0.5/π - 0.5).^2 / 0.3^2) - 0.5
        imshow(φ .> 0, extent=[-1,1,-1,1], cmap="binary")
        title("the domain")
```



```
Out[9]: PyObject <matplotlib.text.Text object at 0x324c33890>
```

This all eventually leads to the following matrix, whose eigenvalues $ $\lambda = \omega 2$arethesquaresofthefrequenciesandwhoseeigenvectorsarethevibratingmodes:$

```
In [10]: x0 = linspace(-1,1,N+2) # includes boundary points, unlike x
         Abox = Laplacian(x0, x0, (x,y) -> 1.0);
         i = find(φ .> 0)
         A = Abox[i,i]
         size(A)

Out[10]: (59779,59779)
```

This is a 60000×60000 matrix, which would too big to even store on my laptop if we stored every entry. Because it is sparse, however, almost all of the entries are zero and we only need to store those.

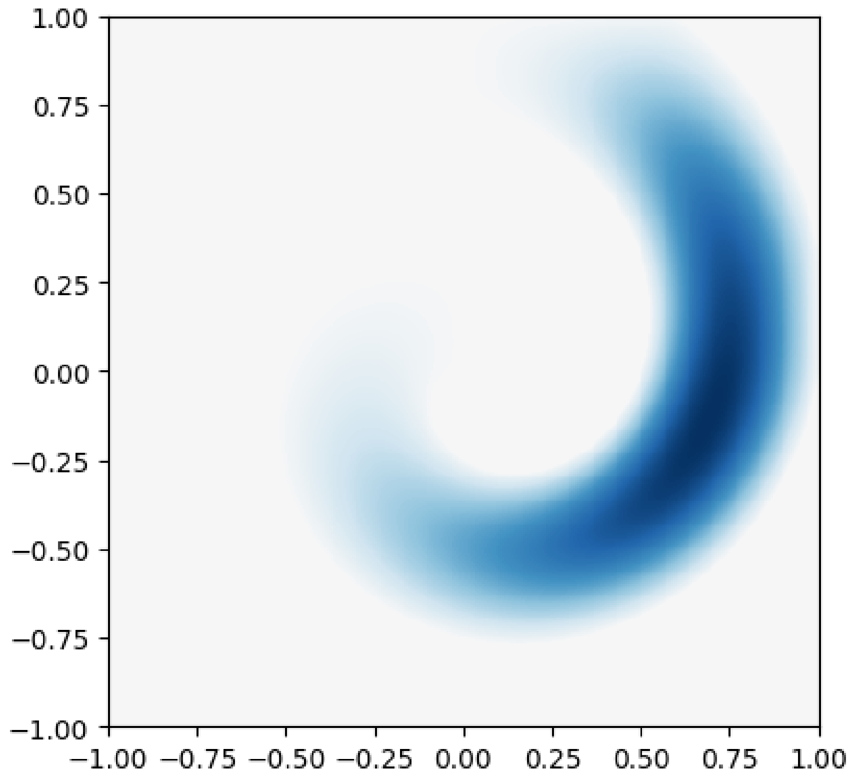
The `nnz` function computes the number of nonzero entries, and we can use it to compute the fraction of nonzero entries in A:

```
In [11]: nnz(A) / length(A)

Out[11]: 8.313882809376213e-5
```

Less than 0.01% of the entries are nonzero! It really pays to take advantage of this.

Now we'll compute a few of the smallest-$|\lambda|$ eigenvectors using `eigs`. We'll use the Interact package to interactively decide which eigenvalue to plot.

```
In [12]: u = zeros(N,N)
         @time λ, X = eigs(A, nev=20, which=:SM);

         f = figure()
         @manipulate for which_eig in slider(1:20, value=1)
             withfig(f) do
                 u[i] = X[:,which_eig]
                 umax = maxabs(u)
                 imshow(u, extent=[-1,1,-1,1], vmin=-umax,vmax=+umax, cmap="RdBu")
             end
         end



Interact.Slider{Int64}(Signal{Int64}(1, nactions=1),"",1,1:20,"horizontal",true,"d",true)


5.075585 seconds (4.33 M allocations: 401.467 MB, 3.69% gc time)

Out[12]:
```



Notice that it took **less than two seconds** to solve for 20 eigenvectors and eigenvalues!

This is because `eigs` is essentially using an algorithm like the power method, that only uses repeated multiplication by A.

Or, sometimes, particularly to find the smallest $|\lambda|$ eigenvectors, it might repeatedly *divide* by A, i.e. solve $Ax = b$ for $b = A^{-1}x$. But it can't actually compute the inverse matrix, and I said that LU factorization was $\sim n^3$ in general. So, what is happening?

4 Sparse-direct solvers for Ax=b

Even if A is a sparse matrix, A^{-1} is generally *not* sparse. However, if you arrange things cleverly, often the L and U factors *are* still sparse!

This leads to something called **sparse-direct solvers**: they solve $Ax = b$ by ordinary Gaussian elimination to find $A = LU$, *but* they take advantage of sparse A to avoid computing with zeros. Moreover, they first re-order the rows and columns of A so that elimination won't introduce too many zeros — this is a tricky problem that mostly involves graph theory, so I won't try to explain it in 18.06.

Julia (and Matlab) both use sparse-direct algorithms automatically when you do `A \ b` if A is stored as a sparse matrix. When they work (i.e. when the L and U factors are sparse), these algorithms are great: fast, memory-efficient, reliable, and worry-free "black boxes".

4.1 Example: Helmholtz solver

The following code solves a **scalar Helmholtz** equation

$$\left[-\nabla^2 - \omega^2\right] u = f(x, y)$$

This equation describes the **propagation of waves u** from a **source f** at a frequency ω is the frequency. For example, imagine water waves travelling across a shallow pond, with $u(x, y)$ being the height of the wave, where f represents an vibrating disturbance that creates the wave.

We discretize this into a matrix equation $Au = f$ by discretizing space (x, y) into a grid and approximating derivatives $-\nabla^2$ by differences on the grid (this is an FDFD method).

Again, don't worry too much about the details of this construction (take 18.303 to find out more). The important thing is that we will have a grid (graph!) of many unknowns, but the problem is sparse because each grid point only "talks" to its 4 nearest neighbors.

```
In [13]: """
         Return `(A,Nx,Ny,x,y)` for the 2d Helmholtz problem.
         """
         function Helmholtz2d(Lx, Ly, ε, ω; dpml=2, resolution=20, Rpml=1e-20)
             # PML σ = σ0 x²/dpml², with σ0 chosen so that
             # the round-trip reflection is Rpml:
             σ0 = -log(Rpml) / (4dpml/3)

             M = round(Int, (Lx+2dpml) * resolution)
             N = round(Int, (Ly+2dpml) * resolution)
             dx = (Lx+2dpml) / (M+1)
             dy = (Ly+2dpml) / (N+1)
             x = (1:M) * dx - Lx/2 - dpml # x grid, centered
             y = (1:N) * dy - Ly/2 - dpml # y grid, centered

             # 1st-derivative matrix
             x′ = ((0:M) + 0.5)*dx # 1st-derivative grid points
             y′ = ((0:N) + 0.5)*dy # 1st-derivative grid points
             ox = ones(M)/dx
             oy = ones(N)/dy
             σx = Float64[ξ < dpml ? σ0*(dpml-ξ)^2 : ξ > Lx+dpml ? σ0*(ξ-(Lx+dpml))^2 : 0.0 for ξ in x
             σy = Float64[ξ < dpml ? σ0*(dpml-ξ)^2 : ξ > Ly+dpml ? σ0*(ξ-(Ly+dpml))^2 : 0.0 for ξ in y
             Dx = spdiagm(1./(1+(im/ω)*σx)) * spdiagm((-ox,ox),(-1,0),M+1,M)
             Dy = spdiagm(1./(1+(im/ω)*σy)) * spdiagm((-oy,oy),(-1,0),N+1,N)
             Ix = speye(M)
             Iy = speye(N)
             return (kron(Ix, Dy.'*Dy) + kron(Dx.'*Dx, Iy) -
                     spdiagm(reshape(Complex128[ω^2 * ε(ξ, ζ) for ζ in y, ξ in x], length(x)*length(y)
                     M, N, x, y)
         end

Out[13]: Helmholtz2d
```

Let's set up a scattering problem with a cylindrical scatterer (a slightly slower wave speed, e.g. a different depth of water, inside a small cylindrical region, with a wavelength $2\pi/\omega$ of 1:

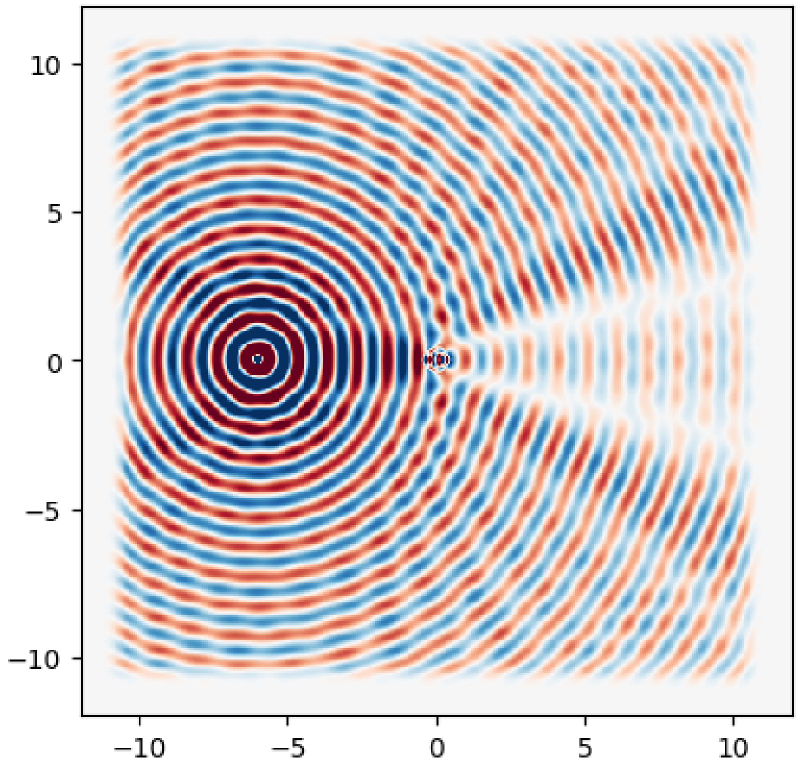
```
In [14]: A, Nx, Ny, x, y = Helmholtz2d(20,20, (x,y) -> hypot(x,y) < 0.5 ? 12 : 1, 2π)
         size(A), nnz(A) / size(A,1)

Out[14]: ((230400,230400),4.991666666666666)
```

Again, this is a huge matrix: 230400×230400. But it is incredibly sparse, so solving it will be no problem:

```
In [15]: nnz(A)/length(A)

Out[15]: 2.166521990740741e-5
```

For the right-hand side `b`, let's use a "point" source on one side. We'll use the **`reshape`** function to convert between 2d arrays and column vectors for solving with `A`. Note that the solution is complex, since it corresponds physically to an oscillating solution $u(x, y)e^{-i\omega t}$ and u has a phase angle due to the absorbing boundary layers (which make `A` non-Hermitian); we'll just plot the real part.

```
In [16]: b = zeros(Nx,Ny)
         b[Nx÷2, Ny÷4] = 1
         @time u = reshape(A \ reshape(b, Nx*Ny), Nx, Ny)
         s = maxabs(real(u)) / 10
         imshow(real(u), cmap="RdBu", vmin=-s, vmax=s,
                extent=(minimum(x),maximum(x),minimum(y),maximum(y)))
```



```
2.925632 seconds (556.26 k allocations: 656.119 MB, 6.38% gc time)
```

```
Out[16]: PyObject <matplotlib.image.AxesImage object at 0x316f63450>
```

We solved a 200000×200000 matrix problem in 2–3 seconds, and less than 1GB of memory. Pretty good!

5 Iterative solvers

Unfortunately, sparse-direct solvers like those we are using above have two limitations:

- They only work **if the matrix is sparse**. There are lots of problems where A has some special structure that lets you compute $A * x$ quickly, e.g. by FFTs, and avoid storing the whole matrix, but for which A is not sparse.

- They **scale poorly** if the sparse matrix comes from a **3d grid or mesh**. For an s-element 1d mesh with $n = s$ degrees of freedom, they have $O(s)$ complexity. For an $s \times s$ 2d mesh with $n = s^2$ degrees of freedom, they take $O(n \log n)$ operations and require $O(n)$ storage. But for a 3d $s \times s \times s$ mesh with $n = s^3$, they take $O(n^2)$ operations and require $O(n^{4/3})$ storage (and you often run out of storage before you run out of time).

The alternative is an **iterative solver**, in which you supply an initial guess for the solution x (often just $x = 0$) and then it *iteratively improves* the guess, converging (hopefully) to the solution $A^{-1}b$, while using *only* matrix-vector operations Ax.

Iterative solvers are the method of choice (or, more accurately, of necessity) for the very largest problems, but they have their downsides. There are *many iterative solver algorithms*, and you have to know a little bit to pick the best one. They *may not converge at all* for non-symmetric A, and in any case may *converge very slowly*, unless you provide a "magic" matrix called a *preconditioner* that is specific to your problem. (It is often a research problem in itself to find a good preconditioner!)

5.1 Toy example: Steepest-descent algorithm

If A is a **real-symmetric positive-definite** matrix, then solving $Ax = b$ is equivalent to minimizing the function:

$$f(x) = x^T A x - x^T b - b^T x$$

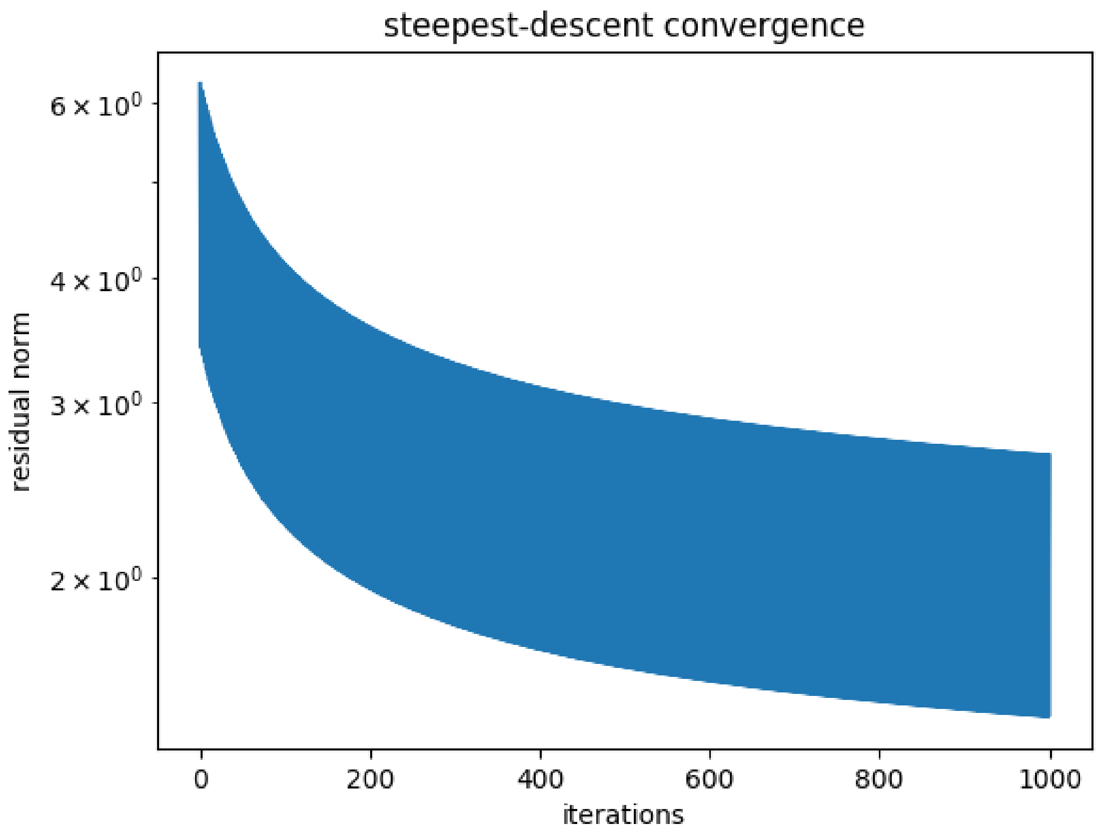
(Just compute $\nabla f = \cdots = Ax - b$, which equals zero at the minimum. The definiteness of A means that the function f is convex, so there is exactly one global minimum.)

One of the simplest iterative algorithms is just to **go downhill**: minimize $f(x + \alpha d)$ over α, where d is the downhill direction $-\nabla f = b - Ax = r$, where r is called the *residual*. We can perform this *line minimization* analytically for this f, for an arbitrary d, to find $\alpha = d^T r / d^T A d$.

The steepest-descent algorithm simply performs this downhill line-minimization repeatedly, starting at an initial guess `x` (typically just `x=0`), e.g. stopping when the norm of the residual is less than some tolerance times the norm of `b`.

```
In [17]: function SD(A, b, x=zeros(b); tol=1e-8, maxiters=1000)
             bnorm = norm(b)
             r = b - A*x # initial residual
             rnorm = [norm(r)] # return the array of residual norms
             Ad = zeros(r) # allocate space for Ad
             for i = 1:maxiters
                 d = r # use the steepest-descent direction
                 A_mul_B!(Ad, A, d) # store matvec A*r in-place in Ar
                 α = dot(d, r) / dot(d, Ad)
                 x .= x .+ α .* d  # in Julia 0.6, this "fuses" into a single in-place update
                 r .= r .- α .* Ad # update the residual (without computing A*x again)
                 push!(rnorm, norm(r))
                 rnorm[end] ≤ tol*bnorm && break # converged
             end
             return x, rnorm
         end

Out[17]: SD (generic function with 2 methods)

In [18]: A = rand(100,100); A = A'*A # a random SPD matrix
         b = rand(100)
         x, rnorm = SD(A, b, maxiters=1000)
         length(rnorm), rnorm[end]/norm(b)

Out[18]: (1001,0.42293442296073136)
```

```
In [19]: semilogy(rnorm)
         title("steepest-descent convergence")
         ylabel("residual norm")
         xlabel("iterations")
```

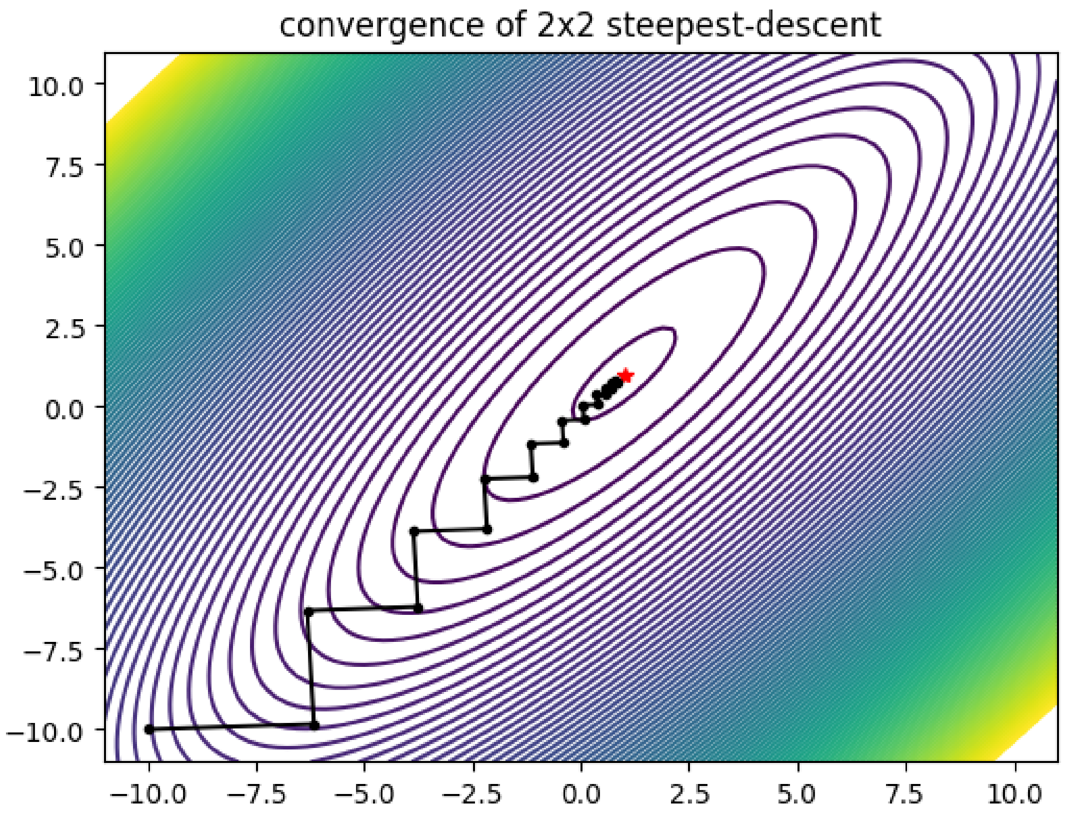


```
Out[19]: PyObject <matplotlib.text.Text object at 0x32669f0d0>
```

To see what's going on, let's try a 2×2 matrix where we can easily visualize the progress.

```
In [20]: θ = 0.9 # chosen to make a nice-looking plot
         Q = [cos(θ) sin(θ); -sin(θ) cos(θ)] # 2x2 rotation by θ
         A = Q * diagm([10,1]) * Q' # a 2x2 matrix with eigenvalues 10,1
         b = A * [1,1] # right-hand side for solution (1,1)
         x1 = linspace(-11,11,100)
         contour(x1', x1, [dot([x1,x2], A*[x1,x2]) - 2*(x1*b[1]+x2*b[2]) for x1 in x1, x2 in x1], level
         plot(1,1, "r*")
         x1s = Float64[]
         x2s = Float64[]
         for i = 0:20
             x, = SD(A, b, [-10.,-10.], maxiters=i)
             push!(x1s, x[1])
             push!(x2s, x[2])
         end
         plot(x2s, x1s, "k.-")
```

```
title("convergence of 2x2 steepest-descent")
```



```
Out[20]: PyObject <matplotlib.text.Text object at 0x323dd6f50>
```

The solution "zig-zags" down the long, narrow valley defined by the quadratic function `f`. This is a common problem of steepest-descent algorithms: they tend to go towards the center of valleys (down the "steep" direction), rather than *along* the valleys towards the solution.

To fix this problem, basically we need to implement some kind of "memory": it has to "remember" that it just "zigged" in order to avoid "zagging" back where it came from.

5.2 From steepest-descent to conjugate-gradient

The most famous way to improve steepest descent with "memory" is the conjugate-gradient algorithm. I won't explain it here (Shewchuk's article is a good introduction to its relationship to steepest descent), but the implementation ends up being *almost identical* to steepest descent. However, instead of setting the line-search direction equal to the downhill direction `r`, the line-search direction is instead a linear combination of `r` with the *previous* search direction:

```
In [21]: function CG(A, b, x=zeros(b); tol=1e-8, maxiters=1000)
             bnorm = norm(b)
             r = b - A*x # initial residual
             rnorm = [norm(r)] # return the array of residual norms
             d = copy(r) # initial direction is just steepest-descent
             Ad = zeros(r) # allocate space for Ad
```

```
            for i = 1:maxiters
                A_mul_B!(Ad, A, d) # store matvec A*r in-place in Ar
                α = dot(d, r) / dot(d, Ad)
                x .= x .+ α .* d  # in Julia 0.6, this "fuses" into a single in-place update
                r .= r .- α .* Ad # update the residual (without computing A*x again)
                push!(rnorm, norm(r))
                d .= r .+ d .* (rnorm[end]/rnorm[end-1])^2 # conjugate direction update
                rnorm[end] ≤ tol*bnorm && break # converged
            end
            return x, rnorm
        end
```

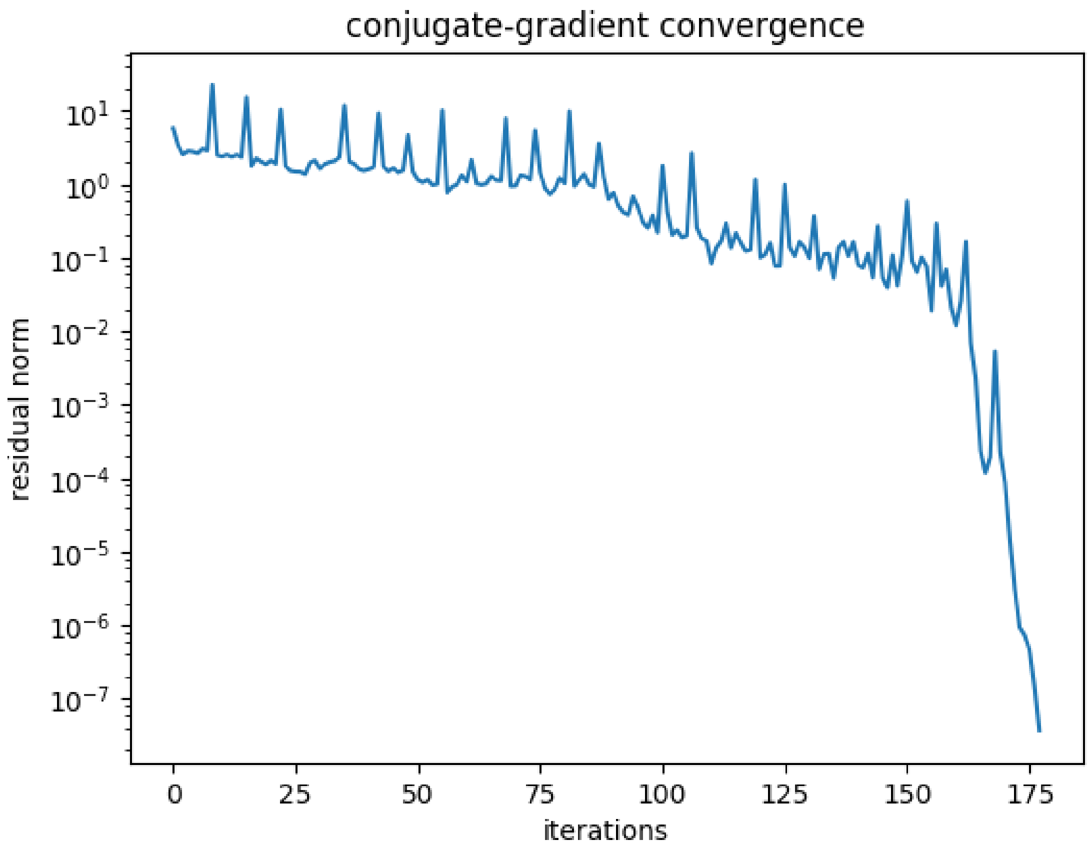
```
Out[21]: CG (generic function with 2 methods)
```

```
In [22]: A = rand(100,100); A = A'*A # a random SPD matrix
         b = rand(100)
         x, rnorm = CG(A, b)
         length(rnorm), rnorm[end]/norm(b)
```

```
Out[22]: (178,6.3442855162928694e-9)
```

After some initial slow progress, the conjugate-gradient algorithm quickly zooms straight to the solution:

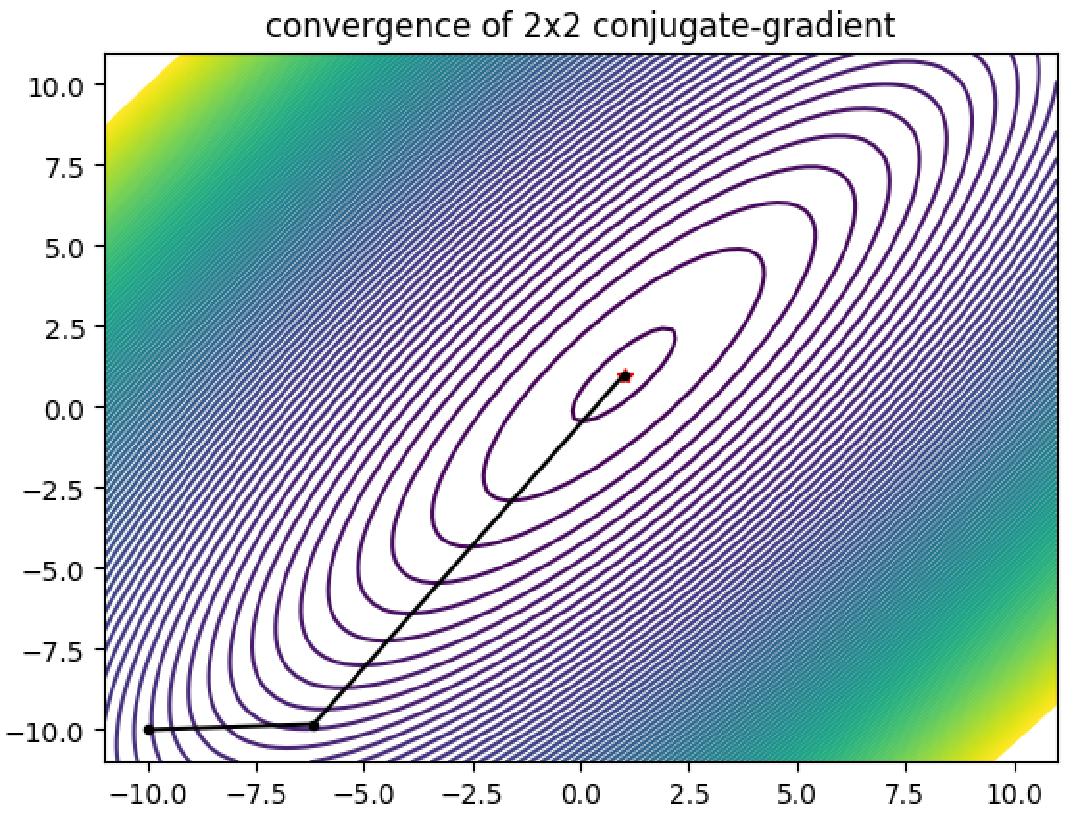
```
In [23]: semilogy(rnorm)
         title("conjugate-gradient convergence")
         ylabel("residual norm")
         xlabel("iterations")
```



```
Out[23]: PyObject <matplotlib.text.Text object at 0x3249f4b10>
```

In our 2x2 problem, you can see that it actually converges in **two steps**:

```
In [24]: θ = 0.9 # chosen to make a nice-looking plot
         Q = [cos(θ) sin(θ); -sin(θ) cos(θ)] # 2x2 rotation by θ
         A = Q * diagm([10,1]) * Q' # a 2x2 matrix with eigenvalues 10,1
         b = A * [1,1] # right-hand side for solution (1,1)
         x1 = linspace(-11,11,100)
         contour(x1', x1, [dot([x1,x2], A*[x1,x2]) - 2*(x1*b[1]+x2*b[2]) for x1 in x1, x2 in x1], level
         plot(1,1, "r*")
         x1s = Float64[]
         x2s = Float64[]
         for i = 0:2
             x, = CG(A, b, [-10.,-10.], maxiters=i)
             push!(x1s, x[1])
             push!(x2s, x[2])
         end
         plot(x2s, x1s, "k.-")

         title("convergence of 2x2 conjugate-gradient")
```



```
Out[24]: PyObject <matplotlib.text.Text object at 0x32510d850>
```

5.3 You don't have to write your own iterative solvers

There are several packages out there with iterative solvers that you can use, e.g. the IterativeSolvers package:

```
In [25]: # do this if you haven't installed it yet: Pkg.add("IterativeSolvers")
         using IterativeSolvers
```

```
WARNING: using IterativeSolvers.rnorm in module Main conflicts with an existing identifier.
```

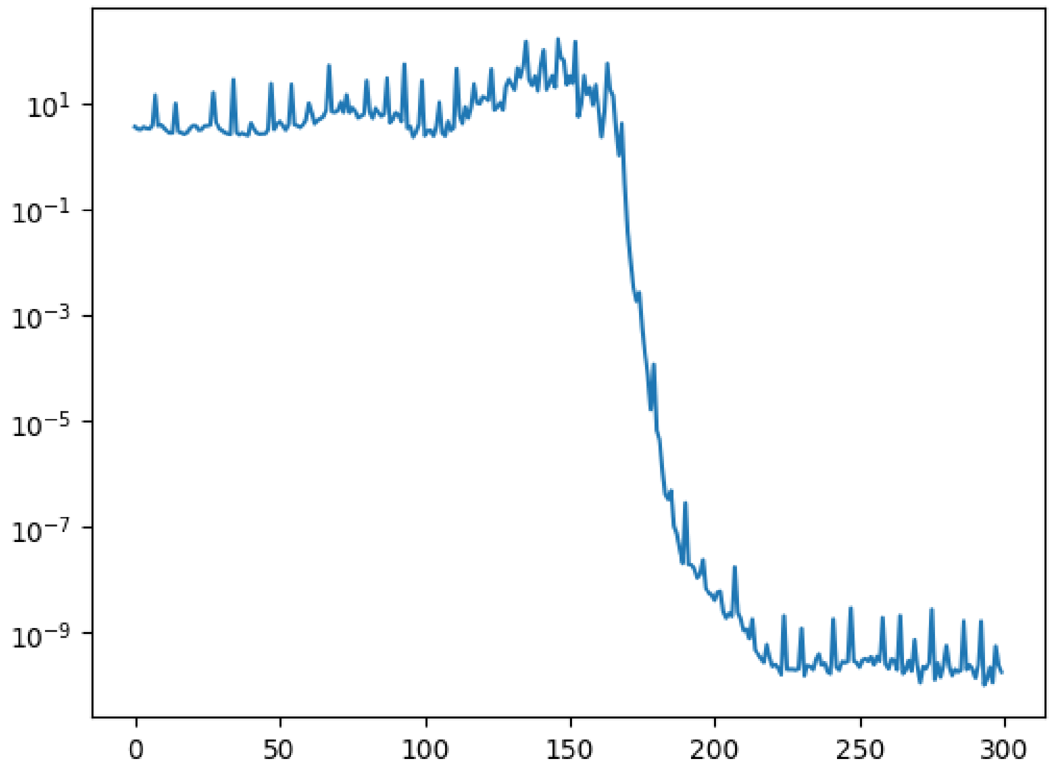
```
In [26]: A = rand(100,100); A = A'*A # a random SPD matrix
         b = rand(100)
         x, ch = cg(A, b, maxiter=300)
         norm(A*x - b) / norm(b)
```

```
Out[26]: 5.000813089046369e-10
```

```
In [27]: dump(ch)
```

```
IterativeSolvers.ConvergenceHistory{Float64,Array{Float64,1}}
  isconverged: Bool false
  threshold: Float64 1.3281146464442924e-13
  mvps: Int64 301
  residuals: Array{Float64}((300,)) [3.54639,3.21832,3.1756,3.48781,3.3264,3.25905,3.75059,14.1867,3.71
```

```
In [28]: semilogy(ch.residuals)
```



```
Out[28]: 1-element Array{Any,1}:
          PyObject <matplotlib.lines.Line2D object at 0x325417d50>
```

5.3.1 PETSc

PETSc.jl provides a Julia interface to the PETSc library: large-scale iterative and sparse solvers for distributed-memory parallel systems. It is a bit harder to use, because it expects you to set up its own kind of sparse matrix that works on distributed-memory systems.

5.4 Preconditioners

Most iterative solvers are *greatly accelerated* if you can provide a *preconditioner*: roughly, an *approximate inverse* of A that is *easy to compute*. The preconditioner is applied at *every step* of the iteration in order to speed up convergence.

For example, let's consider a problem where the matrix $M = L + A$ is a sum of the symmetric-tridiagonal discrete Laplacian L (from above) and a *small, sparse* perturbation A. As our preconditioner, we'll simply use L, since this is a good approximation for M and $L\ b$ is fast (linear time):

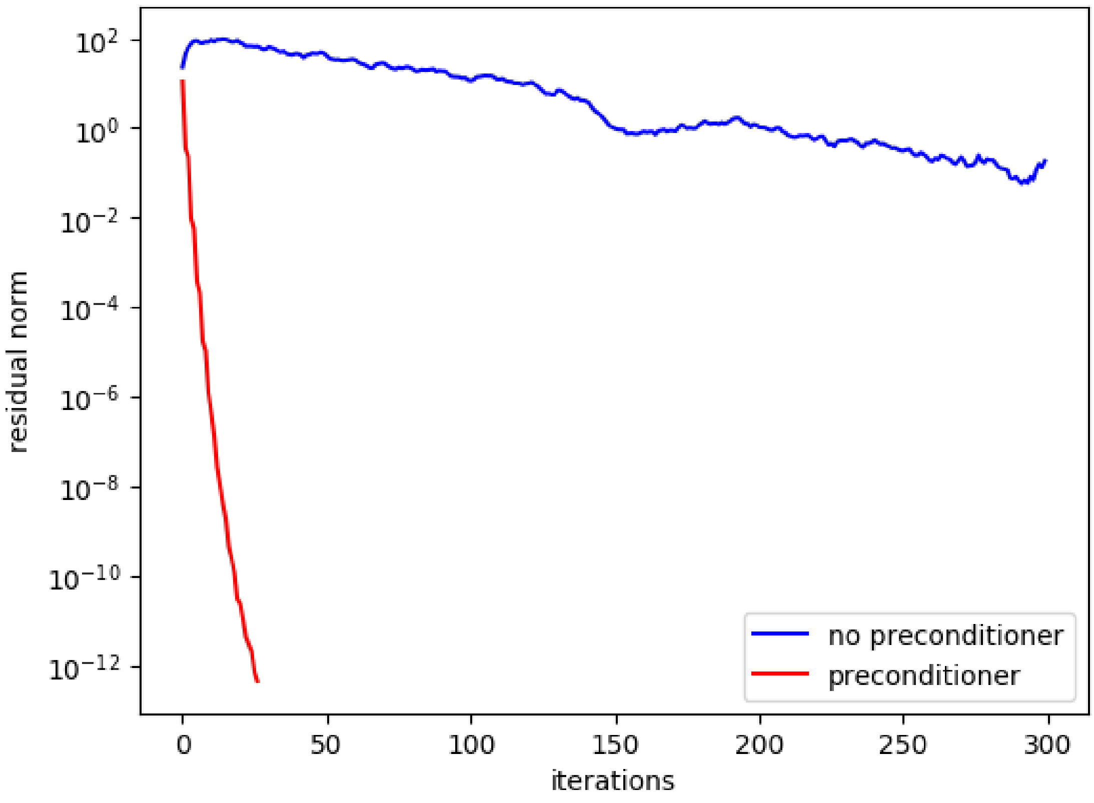
```
In [29]: n = 300
         L = SymTridiagonal(diagm(ones(n)*2) - diagm(ones(n-1),-1) - diagm(ones(n-1), 1))
         b = rand(n)

         A = sprand(n,n,0.001) * 0.001

         M = sparse(L + A)
```

```
Out[29]: 300×300 sparse matrix with 997 Float64 nonzero entries:
                 [1  ,   1]  =  2.0
                 [2  ,   1]  =  -1.0
                 [37 ,   1]  =  0.000413436
                 [1  ,   2]  =  -1.0
                 [2  ,   2]  =  2.0
                 [3  ,   2]  =  -1.0
                 [2  ,   3]  =  -1.0
                 [3  ,   3]  =  2.0
                 [4  ,   3]  =  -1.0
                 [120,   3]  =  0.00049141
                 ⋮
                 [298, 297]  =  -1.0
                 [297, 298]  =  -1.0
                 [298, 298]  =  2.0
                 [299, 298]  =  -1.0
                 [298, 299]  =  -1.0
                 [299, 299]  =  2.0
                 [300, 299]  =  -1.0
                 [40 , 300]  =  0.000192364
                 [86 , 300]  =  0.000916886
                 [299, 300]  =  -1.0
                 [300, 300]  =  2.0
```

Our home-brewed `CG` function above does not accept a preconditioner, but the IterativeSolvers package `cg` function does, and it makes a *huge* difference in the convergence:

```
In [30]: x, ch = cg(M, b, maxiter=300)
         x′, ch′ = cg(M, b, L, maxiter=300)
         semilogy(ch.residuals, "b-")
         semilogy(ch′.residuals, "r-")
         xlabel("iterations")
```

```
ylabel("residual norm")
legend(["no preconditioner", "preconditioner"], loc="lower right")
```



```
Out[30]: PyObject <matplotlib.legend.Legend object at 0x32604c850>
```

If you can find a good preconditioner, you can often speed things up by orders of magnitude. Unfortunately, finding preconditioners is hard and problem-dependent. There are some general "recipes" for things to try, but there are many problems (like the scalar-Helmholtz problem above) where good preconditioners are still an open research problem