Numerical Analysis (CS 450)

4-Credit Hour Project

Due: May 14, 2014 · Progress report due: April 16, 2014 · Out: March 18, 2014

Remarks:

- This project will occasionally refer to files in a *starter kit*. This kit is available as a `.zip` file on the class web page.

Part 1: Use Arnoldi to solve linear systems → GMRES

In this problem, we will be using Arnoldi iteration to (iteratively) solve a linear system of equations $\boldsymbol{Ax} = \boldsymbol{b}$, where $\boldsymbol{A}$ is square, but otherwise general. (i.e. not symmetric or some such)

(a) Consider the following slight variant of the Arnoldi process as shown in Algorithm 4.9 in the textbook:

```
$\boldsymbol{x}_0 = \boldsymbol{b}$          {note use of $\boldsymbol{b}$ as starting vector}
$\boldsymbol{q}_1 = \boldsymbol{x}_0/\|\boldsymbol{x}_0\|_2$
for $k$ in $1, 2, \dots, n$ do
    $\boldsymbol{u}_k = \boldsymbol{A}\boldsymbol{q}_k$
    for $j$ in $1, 2, \dots, k$ do
        $h_{jk} = \boldsymbol{q}_j^T \boldsymbol{u}_k$
        $u_k = \boldsymbol{u}_k - h_{jk}\boldsymbol{q}_j$
    end for
    $h_{k+1,k} = \|\boldsymbol{u}_k\|_2$
    if $h_{k+1,k} = 0$ then
        stop
    end if
    $\boldsymbol{q}_{k+1} = \boldsymbol{u}_k/h_{k+1,k}$
    $(*)$
end for
```

Let $\boldsymbol{Q}_k$ be the matrix consisting of the first k columns of $\boldsymbol{Q}$, and let $\boldsymbol{H}_k$ be the top-left $(k+1)\times k$ submatrix of $\boldsymbol{H}$.

Show that, at the point $(*)$ in the algorithm, at iteration k, the identity $\boldsymbol{Q}_{k+1}^T \boldsymbol{A} \boldsymbol{Q}_k = \boldsymbol{H}_k$ holds.

Does the 'reverse' identity $\boldsymbol{A} = \boldsymbol{Q}_{k+1}\boldsymbol{H}_k\boldsymbol{Q}_k^T$ hold as well?

(b) Recall that the columns of $\boldsymbol{Q}_k$ span the kth Krylov subspace $K_k = \operatorname{span}\{\boldsymbol{b}, \boldsymbol{Ab}, \dots, \boldsymbol{A}^{k-1}\boldsymbol{b}\}$.

Let $\boldsymbol{x}_k \in K_k$ be the vector in K_k that minimizes the residual $\boldsymbol{r}_k = \boldsymbol{A}\boldsymbol{x}_k - \boldsymbol{b}$ in the 2-norm. Let $\boldsymbol{y}_k$ be such that $\boldsymbol{x}_k = \boldsymbol{Q}_k\boldsymbol{y}_k$.

Show that $\|\boldsymbol{r}_k\|_2 = \|(\boldsymbol{H}_k\boldsymbol{y}_k - \|\boldsymbol{b}\|_2\boldsymbol{e}_1)\|_2$, where $\boldsymbol{e}_1$ is the first unit vector $\boldsymbol{e}_1 = (1, 0, \dots, 0)^T$.

(c) Describe a procedure to find $\boldsymbol{x}_k$, using a linear least-squares problem with $\boldsymbol{H}_k$ as a building block.

(continued on next page...)

(d) Implement the Arnoldi procedure above, and at the point $(*)$ in the algorithm, insert code to compute the norm of the residual $\|\boldsymbol{r}_k\|_2$. If $\|\boldsymbol{r}_k\|_2 <$ `tol` (where `tol` is a parameter passed to your function), then compute $\boldsymbol{x}_k$ and return it as your approximate solution to $\boldsymbol{Ax} = \boldsymbol{b}$

For this subproblem, you may use `numpy.linalg.lstsq` to solve the arising least-squares problem.

A more streamlined version of the method we've just derived is known as the *Generalized minimal residual method*, or *GMRES*.

Use the file `gmres.py` in the starter kit as a starting point for your implementation in this and the following part. Your code should pass the tests specified in this file, i.e. yield errors and residuals around machine precision. Include the output of the tests in your writeup.

Note that the parameter `A_func` is a function to compute the matrix-vector product $\boldsymbol{Ax}$ for a given vector $\boldsymbol{x}$. The full matrix $\boldsymbol{A}$ is never passed to your implementation, and in fact the full matrix may not exist anywhere in memory.

For this subproblem, fill in the implementation for `my_gmres_d`.

(e) Realize that $\boldsymbol{H}_k$ is upper Hessenberg, and thus implement code to solve the least squares problem with $\boldsymbol{H}_k$ using Givens rotations.

(Do not build matrices of size $k \times k$ or larger to apply one or more Givens rotations!)

For this subproblem, fill in the implementation for `my_gmres_e`. Your code should no longer call `numpy.linalg.lstsq` or `numpy.linalg.qr` or use other outside help for solving the least-squares problem. You may use a canned routine for back-substitution.

Part 2: Derive an integral equation for a second-order boundary value problem

The goal of this project is to solve the second-order linear (but non-constant-coefficient) boundary value problem ('BVP')

$$\begin{aligned} u'' + p(x)u' + q(x)u &= r(x), && (1)\\ u(a) &= u_a, && (2)\\ u(b) &= u_b && (3) \end{aligned}$$

on the interval $[a, b]$ (with $b > a$).

Let $L := b - a$ and $\tau(x) := 1 - (x - a)/L$. Realize that $\tau(a) = 1$ and $\tau(b) = 0$.

Now, given some (so-called 'density') function φ, let

$$\begin{aligned} u(x) :=\ & \tau(x)u_a + (1 - \tau(x))u_b \\ & + \frac{1}{L}\left((b - x)\int_a^x \varphi(z)(a - z)dz + (x - a)\int_b^x \varphi(z)(b - z)dz\right). \quad (4) \end{aligned}$$

To solve the above boundary value problem, we choose (4) as the *representation* of our solution, based on the new unknown function φ.

(a) Show that $u(a) = u_a$ and $u(b) = u_b$ for any φ.

(continued on next page...)

(b) Show that

$$u'(x) = \frac{1}{L}\left(-u(a) + u(b) - \int_a^x \varphi(z)(a-z)dz + \int_b^x \varphi(z)(b-z)dz\right).$$

Hint: Realize that the variable you're differentiating (x) occurs in the bounds of the integral. To perform the required differentiation, you may use this relationship:

$$\left(\frac{\partial}{\partial x}\int_a^x f(x,z)dz\right)_{x=x_0} = f(x,x_0) + \int_a^{x_0} \frac{\partial}{\partial x} f(x,z)dz.$$

(c) Show that $u''(x) = \varphi(x)$.

(d) Plug u, u' and u'' into (1) and show that φ satisfies the integral equation

$$\varphi(x) + \int_a^b K(x,z)\varphi(z)dz = R(x) \tag{5}$$

with the so-called 'kernel'

$$K(x,z) = \begin{cases} \frac{-p(x)+(b-x)q(x)}{L}(a-z) & z \le x, \\ \frac{p(x)+q(x)(x-a)}{L}(z-b) & z > x, \end{cases}$$

and the right-hand side

$$R(x) = -[q(x)[\tau(x)u_a + (1-\tau(x))u_b] + \frac{p(x)}{L}(-u_a + u_b) - r(x)]. \tag{6}$$

Part 3: Build a BVP solver

In this part, we will be using the integral equation machinery just developed to create a somewhat slow, second-order-accurate solver for our boundary value problem.

(a) Complete the function `apply_kernel` according to its documentation in the file `bvp.py` in the starter kit.

Submit this code in a modified version of `bvp.py`.

(b) Complete the function `solve_bvp` according to its documentation in the file `bvp.py` in the starter kit.

To actually solve the BVP, perform the following steps:

1. Evaluate the right-hand R of the integral equation from (5) on `mesh`. Call the resulting vector $\boldsymbol{R}$.
2. Realize that applying the kernel to a function is a linear operation. Let $\boldsymbol{A}$ be the matrix representing this operation. Note that $\boldsymbol{A}$ is never explicitly built—we can only *apply* $\boldsymbol{A}$ to a vector using `apply_kernel`.

   Use GMRES (as implemented earlier) to solve the integral equation (5), i.e.
   $(\boldsymbol{I} + \boldsymbol{A})\boldsymbol{\varphi} = \boldsymbol{R}$.
   Evaluate $\boldsymbol{A}\boldsymbol{\varphi}$ by a call to `apply_kernel`.

(continued on next page...)

3. Recover the solution vector $\boldsymbol{u}$ from $\boldsymbol{\varphi}$ using (4).

   This may be easiest to do by yet again calling `apply_kernel` with a different `kernel` argument.

Submit this code in a modified version of `bvp.py`.

(c) Your code in `bvp.py` should pass the tests applied by `test_bvp.py` in the starter kit. Your estimated order of accuracy should be 2, or close to it.

Include the output of `test_bvp.py` in your writeup.

(d) Count the number of iterations that GMRES requires to converge to a residual tolerance of 10^{-10}. Make your code output this information.

How does this number of iterations depend on the number of discretization points n? Include a plot of the number of iterations vs. the number of discretization points in your writeup. Choose your range of points such that the largest problem still finishes in about ten seconds.

Part 4: A toolkit for composite high-order discretization

In this problem, we will be building a toolkit for discretizing (and integrating) functions represented as piecewise polynomials.

You should use the file `legendre_discr.py` from the starter kit as a starting point for your implementation.

(a) Begin by writing the constructor of the discretization class. The constructor should compute and set all the attributes documented in the code template. If the other subroutines you write end up requiring precomputation, throwing that precomputation into the constructor is likely also reasonable.

(If you're a bit hazy on object-oriented programming, now is a good time to look up the relevant section of the Python tutorial[1]. If you've never even heard the word, don't be scared. It's neither complicated nor a big deal.)

Call the `scipy` function mentioned in the documentation of the code template to obtain the Gauss-Legendre nodes on the interval $(-1, 1)$ and map them into each of the subintervals.

(b) Next, fill out the `integral` method in the template.

Use the Gaussian quadrature weights that you got from the SciPy routine on each subinterval. (Note that they'll need scaling to reflect the affine mapping of the points.)

A small part of the test script (see below) should now run already—up until the missing functions from the next part are called.

(c) Last, fill out the `left_indefinite_integral` and `right_indefinite_integral` routines according to their documentation.

Note that those ask for a '*spectral integration matrix*'. Suppose $\mathbf{x} := (x_i)$ are the Gauss-Legendre nodes on the interval $(-1, 1)$. Then the spectral integration matrix A for that interval has the following effect:

[1] `http://docs.python.org/2.7/tutorial/classes.html`

(continued on next page...)

$$(Af(\mathbf{x}))_i \approx \int_{-1}^{x_i} f(x)dx$$

This property can help you find the spectral integration matrix, by inverting the matrix equation

$$A\begin{pmatrix} x_0^0 & x_0^1 & \cdots & x_0^n \\ x_1^0 & x_1^1 & \cdots & x_1^n \\ \vdots & \vdots & \ddots & \vdots \\ x_n^0 & x_n^1 & \cdots & x_n^n \end{pmatrix} = \begin{pmatrix} \int_{-1}^{x_0} \xi^0 d\xi & \int_{-1}^{x_0} \xi^1 d\xi & \cdots & \int_{-1}^{x_0} \xi^n d\xi \\ \int_{-1}^{x_1} \xi^0 d\xi & \int_{-1}^{x_1} \xi^1 d\xi & \cdots & \int_{-1}^{x_1} \xi^n d\xi \\ \vdots & \vdots & \ddots & \vdots \\ \int_{-1}^{x_n} \xi^0 d\xi & \int_{-1}^{x_n} \xi^1 d\xi & \cdots & \int_{-1}^{x_n} \xi^n d\xi \end{pmatrix}$$

Note that this method of finding A is not suitable for large n because of poor conditioning of the matrix on the left (called the Vandermonde matrix[2]), but it's good enough for our purposes.

Observe that the spectral integration matrix implements the indefinite integral (a.k.a. 'antiderivative') on each subinterval. You'll have to think about how to connect the antiderivatives on each subinterval so that the routine computes the indefinite integral on the whole domain. This will likely involve using Gaussian quadrature in addition to the spectral integration matrices.

Make sure to keep this routine at linear-time cost as documented.

(d) Run the file `test_legendre_discr.py` from the starter kit. Ensure that the test it performs succeed with your implementation.

Include the output in your writeup.

Part 5: Build a fast and accurate BVP solver

The numerical objective of this part is twofold:

- First, to increase the order of accuracy of your BVP solver from the previous part from 2 to a user-specified number.
- Second, to reduce its computational cost of from $O(n^2)$ to $O(n)$. We will achieve this by reducing the cost of applying the integral operator.

In summary, we will make your earlier code faster and more accurate at the same time.

The solver you built in the previous part had an $O(n^2)$ cost, because there were n integrals to evaluate, each at a cost of $O(n)$. This seemed unavoidable because the kernel changes with every x, so it would seem as though the integral would have to be recomputed for each x. It turns out that this is not the case.

Realize that (each part of) the kernel can be factored according to:

$$K(x,z) = \begin{cases} f_l(x)g_l(z) & z \le x \\ f_r(x)g_r(z) & z > x. \end{cases}$$

As a result, the matrix representing the kernel has rank one, i.e. it is the outer product of two vectors $\boldsymbol{K} = \boldsymbol{v}\boldsymbol{w}^T$. Matrices that have (exactly or approximately) low rank are nice because the

[2]`https://en.wikipedia.org/wiki/Vandermonde_matrix`

(continued on next page...)

matrix-vector product with them can be computed quite cheaply. Suppose $\boldsymbol{\varphi}$ is a vector representing the density, we can use

$$K\boldsymbol{\varphi} = (\boldsymbol{v}\boldsymbol{w}^T)\boldsymbol{\varphi} = \boldsymbol{v}(\boldsymbol{w}^T\boldsymbol{\varphi})$$

to get an $O(n)$ matrix-vector product.

(a) We seek to compute

$$\begin{aligned}&\int_a^b K(x,z)\varphi(z)dz\\ =&\int_a^x K(x,z)\varphi(z)dz + \int_x^b K(x,z)\varphi(z)dz\\ =&f_l(x)\int_a^x g_l(z)\varphi(z)dz + f_r(x)\int_x^b g_r(z)\varphi(z)dz\\ =:&f_l(x)G_l(x) + f_r(x)G_r(x).\end{aligned}$$

Use the above formula to write a linear-time subroutine that computes the effect of the kernel on a function discretized using the toolkit from the previous problem.

Recall that you implemented “left-running” and “right-running” indefinite integrals in your discretization toolkit at $O(n)$ cost.

Complete the function `apply_kernel` according to its documentation in the file `fast_bvp.py` in the starter kit.

Submit this code in a modified version of `fast_bvp.py`.

(b) Adapt your BVP solver from earlier to use the subroutine from the previous subproblem.

Complete the function `solve_bvp` according to its documentation in the file `fast_bvp.py` in the starter kit.

Submit this code in a modified version of `fast_bvp.py`.

(c) Your code in `fast_bvp.py` should pass the tests applied by `test_fast_bvp.py` in the starter kit.

Include the output of `test_fast_bvp.py` in your writeup. Observe that the test script verifies that your solver achieves the full order of the polynomial space.

(d) Count the number of iterations that GMRES requires to converge to a residual tolerance of 10^{-10}. Make your code output this information.

How does this number of iterations depend on the number of discretization points n? Include a plot of the number of iterations vs. the number of discretization points in your writeup. Choose your range of points such that the largest problem still finishes in about ten seconds.

(e) For the final (most complicated) example from the test script, write a script `timing.py` that creates a (properly labeled) plot of computational time vs. the number of subintervals used. Verify that the plot is (approximately) linear, and include it in your writeup.

Choose the number of subintervals as powers of two. Choose the largest such power so that for that subinterval count, your code still runs in 10 seconds or less. (For my code on my laptop at order 5, that’s 2^{18}.)