Integral Equations and Fast Algorithms (CS 598AK)

Homework Set 4

Due: October 17, 2013 · Out: October 3, 2013

Problem 1: Systems of integral equations

Let X_i $(i = 1, \ldots, n)$ be Banach spaces with norms $\|\cdot\|_i$. Show that the product space $X := (X_1 \times \cdots \times X_n)$ of n-tuples $\phi = (\phi_1, \ldots, \phi_n)$ with the maximum norm

$$\|\phi\|_\infty := \max_{i=1,\ldots,n} \|\phi_i\|_i$$

is a normed space. (It is a Banach space, too, but you don't need to show that.)

Show that the composite operator defined by the 'matrix of operators'

$$(A\phi)_i := \sum_{k=1}^{n} A_{ik}\phi_k$$

is compact if and only if each of its components $A_{ik} : X_k \to X_i$ is compact. Use this to state Riesz's theorem for systems of integral equations of the second kind. [patterned after Kress LIE, problem 3.4]

Turn in your answer in a PDF called `problem-1.pdf`.

The numerical objective of this homework set is twofold:

- First, to increase the order of accuracy of your BVP solver from the previous homework 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.

  You verified that empirically on the last homework set that that the number of GMRES iterations does not depend on n for the second-kind operators involved in solving this BVP.

  We will neglect the cost of the linear algebra done inside the GMRES algorithm and only consider the cost of the matrix-vector product (i.e. in our case the application of the integral operator). Consequently, all we need to achieve is reduce the cost of applying the operator to $O(n)$. Then, performing a constant (i.e. n-independent) number of GMRES iterations of an $O(n)$ operator is still $O(n)$ work overall.

In summary, we'll make your code from homework set 3 faster and more accurate at the same time.

We will proceed step by step. First, we will build and test some numerical tools, which we will then use to build the solver.

Problem 2: 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 may use this documented code template to guide your implementation:

`https://gist.github.com/inducer/6804774`

Once you're done with this problem, your implementation should let this test script complete successfully:

`https://gist.github.com/inducer/6804737`

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 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:

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

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.

[1] `http://docs.python.org/2.7/tutorial/classes.html`
[2] `https://en.wikipedia.org/wiki/Vandermonde_matrix`

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.

Problem 3: A fast BVP solver

Recall from the previous homework that the boundary value problem

$$u'' + p(x)u' + q(x)u = r(x), \quad u(a) = u_a, \quad u(b) = u_b \tag{1}$$

can be solved using the integral equation

$$\phi(x) + \int_a^b K(x,z)\phi(z)dz = R(x)$$

with

$$u(x) = \tau(x)u(a) + (1-\tau(x))u(b) + \frac{1}{L}\left((b-x)\int_a^x \phi(z)(a-z)dz + (x-a)\int_b^x \phi(z)(b-z)dz\right) \tag{2}$$

(where $L = b - a$) and

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

and

$$R(x) = -[q(x)[\tau(x)u(a) + (1-\tau(x))u(b)] + \frac{p(x)}{L}(-u(a)+u(b)) - r(x)].$$

where $\tau(x) = 1 - (x-a)/L$.

One key reason why the $O(n^2)$ cost in your previous BVP solver seemed unavoidable is that 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[3] one, i.e. it is the outer product of two vectors $K = \mathbf{v}\mathbf{w}^T$. Matrices that have (exactly or approximately) low rank will play a large role in the fast algorithms part of the class because the matrix-vector product with them can be computed quite cheaply. Suppose $\mathbf{u}$ is a vector representing the density, we can use

$$K\mathbf{u} = (\mathbf{v}\mathbf{w}^T)\mathbf{u} = \mathbf{v}(\mathbf{w}^T\mathbf{u}) = \mathbf{v}(\mathbf{w}\cdot\varphi)$$

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

[3] `https://en.wikipedia.org/wiki/Rank_(linear_algebra)`

a) We seek to compute

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

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

```
def apply_fredholm_kernel(discr, fl, gl, fr, gr, density):
    """
    :arg discr: an instance of
        :class:‘legendre_discr.CompositeLegendreDiscretization‘
    :arg fl,gl,fr,gr: functions of a single argument
    """
    # add code here
```

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

b) Adapt your BVP solver from HW3 to use the subroutine from the previous subproblem. (refer to the provided solutions if needed):

```
def solve_bvp(discr, p, q, r, ua, ub):
    """
    :arg discr: an instance of
        :class:‘legendre_discr.CompositeLegendreDiscretization‘
    """

    # add code here
```

Your BVP solver should allow this test script to complete successfully:

`https://gist.github.com/inducer/6805208`

Observe that the test script verifies that your solver achieves the full order of the polynomial space. The code for this and the preceding part should be in a file called `bvp.py`.

c) 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. Make the script pop up this plot interactively. Verify that the plot is (approximately) linear.

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 2011 laptop at order 5, that's 2^{18}.)

Tangential remark: One key freedom afforded by integral equation methods is that you are mostly free to choose a representation in any way that is convenient. That is to say, the representation formula (2) is not unique. Many other valid representations exist, and some even lead to second-kind equations.

To reiterate from lecture 1, the idea of an integral equation method is to

(a) pick a representation,

(b) plug that representation into the differential equation to be solved,

(c) hope that what comes out at the other end is second-kind, and

(d) solve the thing (in our case, numerically) and be done.

Realize that even though you might not have seen the forest for the trees in homework 3, that's precisely what we've done here.