```
"""A linear algebra library for use with JAX."""

from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import numpy as onp

import jax.lax as lax


# Based on work by phawkins@
# The Cholesky routine is Algorithm 1 from
# Haidar, Azzam, et al. "High-performance Cholesky factorization for GPU-only
# execution." Proceedings of General Purpose GPUs. ACM, 2017.

# TODO(mattjj): implement MAGMA-style algorithms from bartvm@


### Linalg functions


def cholesky(a, block_size=1):
  """An unrolled left-looking Cholesky."""
  return _cholesky(LapaxMatrix(a, block_size)).ndarray


def _cholesky(a):
  """An unrolled left-looking Cholesky on a (possibly blocked) LapaxMatrix."""
  n = a.shape[-1]
  if n == 1:
    return sqrt(a) if a.bs == 1 else _cholesky(a.bview(1))

  def solve(a, b):
    return _solve_triangular_right(a.bview(8), b.bview(8), False, True, True)

  out = full_like(a, 0)
  for i in range(0, n):
    if i > 0:
      a[i:, i] -= out[i:, :i] * out[i, :i].T
    out[i, i] = _cholesky(a[i, i])
```

```
        if i < n - 1:
            out[i:, i] = solve(out[i, i], a[i:, i])
    return out


def solve_triangular(a, b, left_side, lower, trans_a, block_size=1):
    """An unrolled triangular solve."""
    return _solve_triangular_right(LapaxMatrix(a, block_size),
                                   LapaxMatrix(b, block_size),
                                   left_side, lower, trans_a).ndarray


def _solve_triangular_right(a, b, left_side, lower, trans_a):
    """An unrolled right-looking triangular solve on (blocked) LapaxMatrices."""
    n = a.shape[-1]

    def solve(a, b):
        return _solve_triangular_left(a, b, left_side, lower, trans_a)

    if n == 1:
        return solve(a.bview(1), b.bview(1))

    out = full_like(b, 0)
    if lower == trans_a:
        if left_side:
            for i in reversed(range(n)):
                out[i, :] = solve(a[i, i], b[i, :])
                if i > 0:
                    a_col = a[i, :i].T if lower else a[:i, i]
                    b[:i, :] -= a_col * out[i, :]
        else:
            for i in range(n):
                out[:, i] = solve(a[i, i], b[:, i])
                if i < n - 1:
                    a_row = a[i+1:, i].T if lower else a[i, i+1:]
                    b[:, i+1:] -= out[:, i] * a_row
    else:
        if left_side:
            for i in range(n):
                out[i, :] = solve(a[i, i], b[i, :])
                if i < n - 1:
```

```
                a_col = a[i+1:, i] if lower else a[i, i+1:].T
                b[i+1:, :] -= a_col * out[i, :]
        else:
            for i in reversed(range(n)):
                out[:, i:] = solve(a[i, i], b[:, i])
                if i > 0:
                    a_row = a[i, :i] if lower else a[:i, i].T
                    b[:, :i] -= out[:, i] * a_row

    return out


def _solve_triangular_left(a, b, left_side, lower, trans_a):
    """An unrolled left-looking triangular solve on *unblocked* LapaxMatrices."""
    assert a.bs == b.bs == 1
    n = a.shape[-1]
    if n == 1:
        return b / a

    out = full_like(b, 0)
    if lower == trans_a:
        if left_side:
            out[-1, :] = b[-1, :] / a[-1, -1]
            for i in reversed(range(n-1)):
                a_row = a[i+1:, i].T if lower else a[i, i+1:]
                out[i, :] = (b[i, :] - a_row * out[i+1:, :]) / a[i, i]
        else:
            out[:, 0] = b[:, 0] / a[0, 0]
            for i in range(1, n):
                a_col = a[i, :i].T if lower else a[:i, i]
                out[:, i] = (b[:, i] - out[:, :i] * a_col) / a[i, i]
    else:
        if left_side:
            out[0, :] = b[0, :] / a[0, 0]
            for i in range(1, n):
                a_row = a[i, :i] if lower else a[:i, i].T
                out[i, :] = (b[i, :] - a_row * out[:i, :]) / a[i, i]
        else:
            out[:, -1] = b[:, -1] / a[-1, -1]
            for i in reversed(range(n-1)):
                a_col = a[i+1:, i] if lower else a[i, i+1:].T
```

```python
            out[:, i] = (b[:, i] - out[:, i+1:] * a_col) / a[i, i]

    return out


### Convenient internally-used matrix model


def full_like(x, val):
    return LapaxMatrix(lax.full_like(x.ndarray, val), x.bs)


def sqrt(x):
    return LapaxMatrix(lax.pow(x.ndarray, lax.full_like(x.ndarray, 0.5)), x.bs)


def _matrix_transpose(ndarray):
    dims = tuple(range(ndarray.ndim))
    dims = dims[:-2] + (dims[-1], dims[-2])
    return lax.transpose(ndarray, dims)


def _make_infix_op(fun):
    return lambda *args: LapaxMatrix(fun(*(a.ndarray for a in args)), args[0].bs)


class LapaxMatrix(object):
    """A matrix model using LAX functions and tweaked index rules from Numpy."""
    __slots__ = ["ndarray", "bs", "shape"]

    def __init__(self, ndarray, block_size=1):
        self.ndarray = ndarray
        self.bs = block_size
        self.shape = tuple(onp.floor_divide(ndarray.shape, block_size)
                                    + (onp.mod(ndarray.shape, block_size) > 0))

    def __getitem__(self, idx):
        return LapaxMatrix(_matrix_take(self.ndarray, idx, self.bs), block_size=1)

    def __setitem__(self, idx, val):
        self.ndarray = _matrix_put(self.ndarray, idx, val.ndarray, self.bs)
```

```
    def bview(self, block_size):
        return LapaxMatrix(self.ndarray, block_size=block_size)

    __add__ = _make_infix_op(lax.add)
    __sub__ = _make_infix_op(lax.sub)
    __mul__ = _make_infix_op(lax.batch_matmul)
    __div__ = _make_infix_op(lax.div)
    __truediv__ = _make_infix_op(lax.div)
    T = property(_make_infix_op(_matrix_transpose))


# Utility functions for block access of ndarrays


def _canonical_idx(shape, idx_elt, axis, block_size=1):
    """Canonicalize the indexer `idx_elt` to a slice."""
    k = block_size
    block_dim = shape[axis] // k + bool(shape[axis] % k)
    if isinstance(idx_elt, int):
        idx_elt = idx_elt % block_dim
        idx_elt = slice(idx_elt, idx_elt + 1, 1)
    indices = tuple(onp.arange(block_dim)[idx_elt])
    if not indices:
        return slice(0, 0, 1), False    # sliced to size zero
    start, stop_inclusive = indices[0], indices[-1]
    step = 1 if idx_elt.step is None else idx_elt.step
    if k != 1 and step != 1:
        raise TypeError("Non-unit step supported only with block_size=1")
    if step > 0:
        end = min(k * (stop_inclusive + step), shape[axis])
        return slice(k * start, end, step), False
    else:
        end = min(k * (start - step), shape[axis])
        return slice(k * stop_inclusive, end, -step), True


def _matrix_put(ndarray, idx, val, block_size=1):
    """Similar to numpy.put using LAX operations."""
    idx_i, idx_j = idx
    sli, row_rev = _canonical_idx(ndarray.shape, idx_i, -2, block_size)
```

```
    slj, col_rev = _canonical_idx(ndarray.shape, idx_j, -1, block_size)
    if not sli.step == slj.step == 1:
        raise TypeError("Non-unit step not supported in assigment.")

    if row_rev or col_rev:
        val = lax.rev(val, *onp.where([row_rev, col_rev]))

    start_indices = [0] * (ndarray.ndim - 2) + [sli.start, slj.start]
    return lax.dynamic_update_slice(ndarray, val, start_indices)


def _matrix_take(ndarray, idx, block_size=1):
    """Similar to numpy.take using LAX operations."""
    idx_i, idx_j = idx
    sli, row_rev = _canonical_idx(ndarray.shape, idx_i, -2, block_size)
    slj, col_rev = _canonical_idx(ndarray.shape, idx_j, -1, block_size)

    start_indices = [0] * (ndarray.ndim - 2) + [sli.start, slj.start]
    limit_indices = list(ndarray.shape[:-2]) + [sli.stop, slj.stop]
    strides = [1] * (ndarray.ndim - 2) + [sli.step, slj.step]
    out = lax.slice(ndarray, start_indices, limit_indices, strides)

    if row_rev or col_rev:
        out = lax.rev(out, *onp.where([row_rev, col_rev]))
    return out
```