18.335 final projects

The final project will be an 8–15 page paper (single-column, single-spaced, ideally using the style template from the *SIAM Journal on Numerical Analysis*), reviewing some interesting numerical algorithm not covered in the course. [Since this is not a numerical PDE course, the algorithm should *not* be an algorithm for turning PDEs into finite/discretized systems; however, your project *may* take a PDE discretization as a given "black box" and look at some other aspect of the problem, e.g. iterative solvers.] Your paper should be written for an audience of your peers in the class, and should include example numerical results (by you) from application to a realistic problem (small-scale is fine), discussion of accuracy and performance characteristics (both theoretical and experimental), and a fair comparison to at **least one competing algorithm** for the same problem. Like any review paper, you should *thoroughly reference* the published literature (citing both original articles and authoritative reviews/books where appropriate [rarely web pages]), tracing the historical development of the ideas and giving the reader pointers on where to go for more information and related work and later refinements, with references cited throughout the text (enough to make it clear what references go with what results). (**Note:** you may re-use diagrams from other sources, but all such usage must be *explicitly credited*, not doing so is plagiarism.) Model your paper on academic review articles (e.g. read *SIAM Review* and similar journals for examples).

A good final project will include:

- An extensive introduction and bibliography putting the algorithm in context. Where did it come from, and what motivated its development? Where is it commonly used (if anywhere)? What are the main competing algorithms? Were any variants of the algorithm proposed later? What do other authors say about it?
- A clear description of the algorithm, with a summary of its derivation and key properties. Don't copy long mathematical derivations or proofs from other sources, but do *summarize* the key ideas and results in the literature.
- A convincing validation on a representative/quasi-realistic test problem (i.e. show that your code works), along with an informative comparison to important competing algorithms. For someone who is thinking about using the algorithm, you should strive to give them *useful* guidance on how the algorithm compares to competing algorithms: when/where should you consider using it (if ever)? Almost never rely on actual timing results — see below!
    - Note that this should be your *own* independent re-implementation of the algorithm (in any language of your choice) based on the mathematical description in the paper(s). (Not based on other people's code.) The point is to demonstrate that you understand the algorithm well enough to implement it correctly, and that you understand how to make useful comparisons between algorithms.

What to submit

You should submit:

1. A final-project proposal by midway through the term, to get feedback on your project topic and other plans.
2. A PDF of your paper (in SIAM format or similar, see above), due on the last day of the term.
3. A `.zip` or `.tar.gz` / `.tgz` archive containing your source code (in whatever programming language). A `README` file briefly outlining what is in each file would also be appreciated. (You don't need to include software packages downloaded from elsewhere and used unmodified, but a note in the `README` about software requirements would be good.)

Frequently asked questions

Frequently asked questions about the final project:

1. *Does it have to be about numerical linear algebra?* No. It can be any numerical topic (basically, anything where you are computing a conceptually real result, not integer computations), excluding algorithms for discretizing PDEs.
2. *Can I use a matrix from a discretized PDE?* Yes. You can take a matrix from the PDE as input and then talk about iterative methods to solve it, etcetera. I just don't want the paper to be about the PDE discretization technique itself.
3. *How formal is the proposal?* Very informal—one page describing what you plan to do, with a couple of references that you are using as starting points. Basically, the proposal is just so that I can verify that what you are planning is reasonable and to give you some early feedback.
4. *How much code do I need to write?* A typical project (there may be exceptions) will include a working proof-of-concept implementation, e.g. in Julia or Python or Matlab, that you wrote to demonstrate that you understand how the algorithm works. Your code does *not* have to be competitive with "serious" implementations, and I encourage you to download and try out existing "serious" implementations (where available) for any large-scale testing and comparisons.
5. *How should I do performance comparisons?* Be very cautious about timing measurements: unless you are measuring highly optimized code or only care about orders of magnitude, timing measurements are more about implementation quality than algorithms. Better to *measure something implementation-independent* (like flop counts, or matrix-vector multiplies for iterative algorithms, or function evaluations for integrators/optimizers), even though such measures have their own weaknesses.

Proposal

During the semester you will submit a 1–2 page **proposal** for your intended project. This is just intendeded so that I can give you early feedback on your plans. Note in particular that I do **not** expect "research" projects; your project should mostly consist of *known* results that you review, re-implement, validate, compare, and otherwise synthesize.

Your proposal should include:

- A *brief* (1–2 paragraph) description of the algorithm and the problem it solves.
- How you intend to *validate* your implementation (e.g. test problems).
- Information on your planned implementation — language, useful libraries you will exploit?
- What algorithm(s) you intend to compare your implementation to.
- How will you compare algorithms? (As noted in the course README, I would recommend against measures like execution time that depend strongly on how well-optimized your implementation is.)
- A few references that you will use as starting points.

Project topics

As described above you have broad flexibility in choosing a project. Some key constraints are:

- Must be "numerical" in the sense of taking real (or complex) numbers in and real numbers out; no integer-only algorithms.
- Must not be covered in class. (This includes topics that *will* be covered; you can look ahead in the web page from previous years to get a sense of what will be covered, and I will also give you feedback if you propose a topic that I plan to cover in class.)
- Must not be about how to discretize a PDE or integral equation. (You can, however, take a discretized PDE as an *input* and talk about techniques to solve it.)
- Must involve nontrivial numerics; a rule of thumb is that it should take at least 100-200 lines of code to implement. (As opposed to algorithms where the analysis is very complicated but the resulting algorithm is trivial to implement.) (You can implement in any language you want, though I would tend to recommend a high-level language like Julia, Python, or Matlab, rather than a low-level language like C or C++.)

A few examples of project topics from past terms are:

- Simultaneous diagonalization of commuting matrices (e.g. this method)
- Nonlinear eigenproblems (e.g. contour-integration methods likeFEAST or other methods like in NEP-PACK).
- Multidimensional cubature with sparse grids (e.g. Smolyak algorithms)
- Fast algorithms for Gaussian quadrature (e.g. these references). Gauss–Patterson rules and other quadrature rules (e.g. quadrature methods for singular integrands or highly oscillatory integrands).
- Monte-Carlo integration with adaptive importance sampling
- Iterative linear-algebra algorithms not covered in class, e.g. BiCGSTAB(ℓ), DQGMRES, or the Jacobi-Davidson eigenproblem algorithm.
- Matrix exponentials: computing the matrix e^A, or maybe just y=e^Ax iteratively where A is sparse (e.g. Krylov methods).
- Discrete cosine transform algorithms (e.g. for audio compression), and other fast transforms (e.g. nonuniform FFTs, spherical-harmonic transforms, etc.)
- Nonlinear optimization algorithms, e.g. ORBIT, VNS, multistart algorithms like MLSL, globally convergent Nelder-Mead, …
- Optimization algorithms specifically for least-square fitting or regularized fitting problems.
- Linear Programming (LP) algorithms (e.g. interior-point or simplex). (My inclination is to *discourage* LP projects, partly because I've seen to many, partly because interior-point methods are very tricky to implement, and mainly because the "fair comparison" part of the project is very difficult.)
- Quasi-Newton methods for nonlinear systems of equations (e.g. Broyden updates)
- Rational approximation (e.g. the Remez algorithm)
- The Fast Multipole Method for the n-body problem
- Uncertainy quantification (UQ) algorithms
- Numerical continuation algorithms (e.g. pseudo arc-length continuation)
- Exactly rounded summation via adaptive-precision arithmetic, e.g. using the algorithms by Neal (callable from Julia) or Shewchuk (used for Python's `fsum`).

(Try leafing through a book on numerical algorithms, e.g. Numerical Recipes, for ideas.)