Numerical Libraries for Scientific Computing

April 1, 2015

Ge Baolai
SHARCNET
Faculty of Science
Western University

Topics

- Numerical libraries for scientific computing
- Numerical libraries at SHARCNET
- Examples
  - Linear algebra
  - FFTW
  - ODE solvers
  - Optimization
  - GSL
  - Multiprecision with MPFUN
  - PETSc (PDE solvers)
- Q&A

Overview

Numerical Libraries

Numerical Computing

- Linear algebra
- Nonlinear equations
- Optimization
- Interpolation/Approximation
- Integration and differentiation
- Solving ODEs
- Solving PDEs
- FFT
- Random numbers and stochastic simulations
- Special functions

Numerical Libraries

More fundamental problems:

- Linear algebra
- Nonlinear equations
- Numerical integration
- ODE
- FFT
- Random numbers
- Special functions

Numerical Libraries

Top Ten Algorithms for Science (Jack Dongarra 2000)

1. Metropolis Algorithm for Monte Carlo
2. Simplex Method for Linear Programming
3. **Krylov Subspace** Iteration Methods
4. The Decompositional Approach to Matrix Computations
5. The Fortran Optimizing Compiler
6. **QR** Algorithm for Computing Eigenvalues
7. Quicksort Algorithm for Sorting
8. **Fast Fourier Transform**
9. Integer Relation Detection
10. Fast Multipole Method

Linear Algebra

Common Linear Algebra Operations

- Basic linear algebra operations
- Solving linear systems $Ax = b$ via **LU** decomposition.
- Solving linear spd systems $Ax = b, A$ spd, via Cholesky factorization.
- Solving least squares problem $\min \|b - Ax\|_2$ via **QR** algorithms.
- Solving eigenvalue problems $Ax = \lambda x$ via QR factorization.
- Solving generalized eigenvalue problems $Ax = \lambda Bx$.
- Calculating **singular value decomposition** $A = U\Sigma V^T$ via QR factorization.
- Solving linear systems via *iterative* methods.
- **Krylov** subspace methods (**KSM**), e.g. CG (converge guaranteed), BCG, QMR, GMRES, BiCGstab, CGS, etc. alone, might be slow.
- Commonly used along with **preconditioning**: $M^{-1}Ax = M^{-1}b$ as finding an approximate inverse of the original matrix. KSM with preconditioning is referred to **KSP** in PETSc.

Numerical Libraries

Most commonly used **linear algebra libraries**

- **LINPACK** (1970s), succeeded by LAPACK.
- **EISPACK** (1970s), succeeded by LAPACK.
- **LAPACK** – Free Fortran subroutines, some C/C++ APIs available, variants:
  - ATLAS – automatically tuned linear algebra systems.
  - **ACML** (**AMD**).
  - CXML (**Compaq** diseased), ESSL (**IBM**), SCSL (**SGI**), etc.
  - **IMSL** (40+ old, now owned by **Rogue Waves**).
  - MLIB (**HP**).
  - **MKL** (**Intel**).
  - **NAG** (**Numerical Algorithms Group**), etc.
- **SuperLU** – Sparse direct solvers.
- **ScaLAPACK** – LAPACK for distributed systems.
- **SPAI** – Sparse approximate inverse.
- **PETSc + SLEPc** – iterative solvers, eigen solvers.
- See **@netlib** http://www.netlib.org/ for more.

LAPACK

- BLAS level 1, 2 and 3 routines for basic linear algebra operations. The most famous one and studied extensively for high performance computing is matrix-matrix multiplication **DGEMM**. A famous name associated with DGEMM is **Goto Kazushige 後藤 和茂** , a Japanese researcher who devoted his life to making the world's fastest matrix-matrix multiplication code, used to benchmark world's fastest computers. Goto used ship stand alone DGEMM routines, often known as Goto library, best tuned for the target hardware, usually beats any other implementations!
- Drivers for solving various problems.
- In addition to math operands, most routines has auxiliary tuning parameters that one can fine tune to achieve the optimal performance on the target hardware.
- Freely available in Linux distributions.
- Originally written in Fortran 77, callable from Fortran 90, C/C++. C/C++ interfaces are available.
- For dense matrices only, not for sparse matrices.
- Does not include iterative solvers.

Linear algebra libraries (cont'd)

- **ARPACK** – A collection of Fortran 77 subroutines for solving large scale eigenvalue problems using **AR**noldi methods.
  - Allow to compute a few eigenvalues with users specified features.
  - Underlying solver used by MATLAB, Octave, Scilib and R, etc.
  - It can use any matrix storage format, with matrix operations done through ***"reverse communication interface"*** (RCI) to the calling procedure.
  - **ARPACK++** - C++ interface to ARPACK; **PARPACK** – Parallel ARPACK.
  - Minor modifications needed to install.

  Alternatives:
  - **BLOPEX**
  - **PRIMME** – See a survey on eigenvalue solvers (Andreas Stathopoulos, 2006)
  - **PETSc+SLEPc**
- **FEAST** – Fast, high performance eigensolver, already integrated into intel MKL 11.x
  - Includes both ready to use interfaces and RCI interfaces
  - Can compute all eigenvalues and eigenvectors for given search interval

Linear algebra libraries (cont'd)

- **PARDISO** – Package for solving dense and sparse linear systems.
- **Aztec** – For solving large linear sparse systems.
- **MUMPS** – A massive parallel direct sparse solver.
- **Hypre** - A library for solving large, sparse linear systems, using common **Krylov** subspace based methods. Contains a suite of **preconditioners**.

See also collections, e.g.:

- Intel MKL
- GSL
- Boost.Numerics
- Apache Commons Math (mostly java?)

Example: Solving linear system $Ax = b$ using LU decomposition. Let

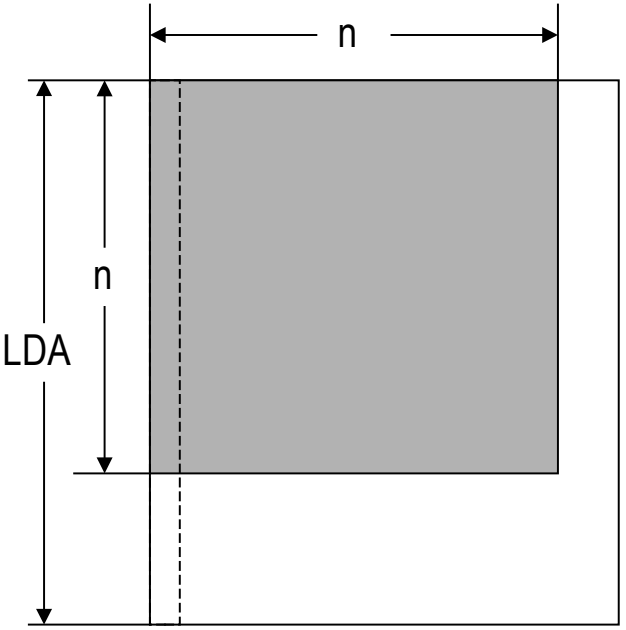
$$A = LU$$

then $LUx = b$, we solve

$$\begin{aligned} Ly &= b, \\ Ux &= y. \end{aligned}$$

This can be all done by one LAPACK subroutine **xGESV**

call dgesv(n, num_rhs, A, LDA, ipiv, x, LDA, info)



Example: (Cont'd) A by-product, we can compute the determinant of a matrix. For an n-by-n matrix, the determinant is defined as

$$\det(A) = \sum_{j_1, j_2, \ldots, j_n} (-1)^{\sigma(j_1, j_2, \ldots, j_n)} a_{ij_1} a_{ij_2} \cdots a_{ij_n}$$

where $\sigma(j_1, j_2, \ldots, j_n)$ denotes the permutation of column indices.

Alternatively, via LU decomposition $A = PLU$

$$\begin{aligned} \det(A) &= \det(P)\det(L)\det(U) \\ &= (-1)^{\sigma} \det(U) \end{aligned}$$

This can be done with LAPACK subroutine **xGETRF** (with ***x*** = ***d*** for double precision)

call dgetrf(n, n, A, LDA, ipiv, info)

The permuation of rows is stored in ipiv: ipiv[i] containts the index of row that is swapped with row i.

Examples

Example: Compute the eigenvalues $Ax = \lambda x$. Wilkinson test matrix: An n-by-n tridiagonal matrix, with diagonal elements

$$\frac{n-1}{2}, \frac{n-3}{2}, \frac{n-5}{n}, \ldots\ldots, \frac{n-5}{n}, \frac{n-3}{2}, \frac{n-1}{2}$$

and 1's on off diagonals. The Wilkinson matrix of order seven looks as follows

$$A = \begin{bmatrix} 3 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 2 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 2 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 3 \end{bmatrix}.$$

The Wilkinson test matrix has a property that it has seeming the same eigenvalues in pairs, but they are really not the same. This may lead to the demand for higher precisions.

Examples

Example: Wilkinson matrix (cont'd). The LAPACK routine to use is

```
call dstev('n', N, D, T, Z, DMAX, work, info)
```

On entry

- D – Vector containing the diagonal elements.
- T – Vector containing the off diagonal elements.

On return

- D – Contains eigenvalues.
- Z – Contains eigenvectors.

Examples

Example: Compute selected eigenvalues and eigenvectors of generalized **symmetric positive definite** eigenproblems

(1) $$Ax = \lambda Bx$$

This can be done by the LAPACK subroutine **xSPGVX**, e.g.

```
call dspgvx( 1, 'N', 'I', 'U', n, A, B, VL, VU, iL, iU, 1, atol, &
   num_eigs, eigs, eigv, 1, work, iwork, ifail, info )
```

But alternatively, we can separate the solution to (1) into two steps. Let $B = LU$, then we can write

$$Ax = \lambda LU$$

or

$$\Rightarrow L^{-1}Ax = \lambda Ux \quad \Rightarrow L^{-1}AU^{-1}Ux = \lambda Ux$$

$$\begin{aligned} L^{-1}AU^{-1}y &= \lambda y, \\ Ux &= y. \end{aligned}$$

Examples

Remarks:

- The original generalized eigenvalue problem is separated into two problems: Part 1: LU (or Cholesky for *spd* matrices) factorization and Part 2: Solving a regular eigenvalue problem.
- Profiling the implementation of Part 1 and Part 2 shows 55+% vs. 45-% on execution time
- *"…It does the transformation from the Generalized Eigenvalue Problem to the Standard Eigenvalue Problem, using a new DAG algorithm and OpenMP. The Speed up for a 7000 by 7000 matrix is 1000 ! Yes 1000 times faster compared to the EISPACK routines which cannot be parallelized because of the data dependencies in the Cholesky algorithm."* – Nick Chepurniy (former SHARCNET staff).

Multiprecision Packages

Numerical Libraries

Example: Compute the eigenvalues of Wilkinson test matrix with higher than IEEE double precision.

The Wilkinson test matrix has a property that it has seemingly the same eigenvalues in pairs.

Really? This is where we need higher or arbitrary precisions.

For arbitrary precisions, two packages to consider:

- **GMP** (C/C++ interface)
- **MPFUN** (Fortran) or **ARPREC** (C++ implementation with C/C++ and Fortran interfaces)

```
-0.96413217269049067
 0.21266284251310127
  1.0781644067346108
  1.6547300922541341
  2.4007717515805513
  2.6137235587096592
  3.4862749555221253
  3.5176393110600825
  4.4989686209327697
  4.5011924696537475
  5.4999538446482141
  5.5000502386255690
  6.4999986232779925
  6.5000014559541404
  7.4999999704788101
  7.5000000307300576
  8.4999999995215454
  8.5000000004932090
  9.4999999999939302
  9.5000000000062279
  10.499999999999938
  10.500000000000062
  11.500000000000004
  11.500000000000012
  12.499999999999995
  12.500000000000009
  13.499999999999998
  13.500000000000004
  14.499999999999998
  14.500000000000007
  15.500000000000616
  15.500000000000622
  16.500000000054481
  16.500000000054488
  17.500000003808111
  17.500000003808125
  18.500000205070439
  18.500000205070439
  19.500008158672937
  19.500008158672948
  20.500225680185167
  20.500225680185167
  21.503952002665365
  21.503952002665386
  22.538941119306436
  22.538941119306440
  23.710678647333044
  23.710678647333058
  25.246194182903331
  25.246194182903359
```

Examples

Fortran code using MPFUN

```
program steig
  use mpmodule
  implicit none
  type ( mp_real ), allocatable :: d(:), t(:), work(:), z(:,:)
  integer :: info, integer :: i, n, m, num_digits

  print *, 'Enter matrix size:'
  read *, n
  print *, 'Enter accuracy (number of digits):'
  read *, num_digits
  call mpinit( num_digits )
  call mpsetoutputprec( num_digits )

  allocate(d(n), t(n), z(n,n), work(2*n-2))

  create Wilkinson eigenvalue test matrix

  call imtql1(n, d, t, info)
  print *, 'Eigenvalues (Using EISPACK):'
  do i = 1, n
    call mpwrite( 6,c(i) )
  end do
end program steig
```

EISPACK subroutine

```
! This QL algorithm was written in the 1969s…
! We don't want to rewrite it!
      subroutine imtql1(n,d,e,ierr)
      use mpmodule
      integer i,j,l,m,n,ii,mml,ierr
      type ( mp_real ) d(n),e(n)
      type ( mp_real ) b,c,f,g,p,r,s,tst1,tst2
c
c this subroutine is a translation of the algol procedure
c imtql1, num. math. 12, 377-383(1968) by martin and
c wilkinson,
c … …

      … … Rest of the code … …
      … …
      return
      end
```

With little effort – ***three lines only*** in this case – I was able to compute eigenvalues to arbitrary digits!

This example also shows the simplicity of Fortran superior to other languages for scientific computing.

Examples

Example: Computing eigenvalues of Wilkinson matrix of order n using double (15) and 40 digits

-0.96413217269049067	10 ^	-1 x -9.64132172690478846920380765355872527282010249478 5,
0.21266284251310127	10 ^	-1 x 2.12662842513110729570212849445910220489845435747 5,
1.0781644067346108	10 ^	0 x 1.07816440673461058675152726093351525017534624259 7,
1.6547300922541341	10 ^	0 x 1.65473009225413448557976675325788614177928533334 8,
2.4007717515805513	10 ^	0 x 2.40077175158054965347171945367374512583151265893 6,
2.6137235587096592	10 ^	0 x 2.61372355870965797025170541936697335498150958418 1,
3.4862749555221253	10 ^	0 x 3.48627495552212737731613134621904278290628307855 2,
3.5176393110600825	10 ^	0 x 3.51763931106008520691335774681901182199739170660 8,
4.4989686209327697	10 ^	0 x 4.49896862093276935220856516401688357692432379521 9,
4.5011924696537475	10 ^	0 x 4.50119246965374796706003080014237906774702964229 6,
5.4999538446482141	10 ^	0 x 5.49995384464821942883822982203918054488467238050 7,
5.5000502386255690	10 ^	0 x 5.50005023862557449361462920243145316463521213446 6,
6.4999986232779925	10 ^	0 x 6.49999862327798891559199591314769397456191620407 7,
6.5000014559541404	10 ^	0 x 6.50000145595413679989719233398395137782707880107 7,
7.4999999704788101	10 ^	0 x 7.49999997047881137165744688031253514500963093055 9,
7.5000000307300576	10 ^	0 x 7.50000003073005952645705174835347834575943254645 7,
8.4999999995215454	10 ^	0 x 8.49999999952154092797877180712946806525424136729 5,
8.5000000004932090	10 ^	0 x 8.50000000493206277760349451609484527161829931395,
9.4999999999939302	10 ^	0 x 9.49999999993923902705172850882166649743950242226,
9.5000000000062279	10 ^	0 x 9.50000000006222657447847663151767192196337560297,
10.499999999999938	10 ^	1 x 1.04999999999999378571957218511428522489549239463 5,
10.500000000000062	10 ^	1 x 1.05000000000000633502302968963886515973645900316 4,
11.500000000000004	10 ^	1 x 1.14999999999999994769172892605335894662387106957 9,
11.500000000000012	10 ^	1 x 1.15000000000000005314587972139552209177656679226 8,
12.499999999999995	10 ^	1 x 1.24999999999999999965986842447218576604402987635 9,
12.500000000000009	10 ^	1 x 1.25000000000000000402493853651720306437850545540,
13.499999999999998	10 ^	1 x 1.35000000000000000441699670997498582393493325464 2,
13.500000000000004	10 ^	1 x 1.35000000000000000442144353253788810577961989052 7,
14.499999999999998	10 ^	1 x 1.45000000000000057424636181085647939243013823437 2,
14.500000000000007	10 ^	1 x 1.45000000000000057424638473249757452750886745169 7,
15.500000000000616	10 ^	1 x 1.55000000000006197895720929520873154825160583877 7,
15.500000000000622	10 ^	1 x 1.55000000000006197895720939799597532956911737456 1,
16.500000000054481	10 ^	1 x 1.65000000000544881079904291432979229450971073752 8,
16.500000000054488	10 ^	1 x 1.65000000000544881079904291473446541392706013251 2,
17.500000003808111	10 ^	1 x 1.75000000038081268835369405683888103095339257475 9,
17.500000003808125	10 ^	1 x 1.75000000038081268835369405684029103912374083887 7,
18.500000205070439	10 ^	1 x 1.85000002050704378003186498845163095435240885403 9,
18.500000205070439	10 ^	1 x 1.85000002050704378003186498845163533325185386156 5,
19.500008158672937	10 ^	1 x 1.95000081586729450104640503452370491581316586454 1,
19.500008158672948	10 ^	1 x 1.95000081586729450104640503452370492801040845345 3,
20.500225680185167	**10 ^**	**1 x 2.050022568018517034412527273640041808200744223750,**
20.500225680185167	**10 ^**	**1 x 2.050022568018517034412527273640041808231370908819,**
21.503952002665365	10 ^	1 x 2.15039520026653613283661145681180694964160680482 1,
21.503952002665386	10 ^	1 x 2.15039520026653613283661145681180694964167564557 0,
22.538941119306436	10 ^	1 x 2.25389411193064088976740590263711884819586263287,
22.538941119306440	10 ^	1 x 2.25389411193064088976740590263711884819586491979,
23.710678647333044	10 ^	1 x 2.37106786473330464883276996339336202669720139832 2,
23.710678647333058	10 ^	1 x 2.37106786473330464883276996339336202669720189606 3,
25.246194182903331	10 ^	1 x 2.52461941829033575705868839676720557403112561189 5,
25.246194182903359	10 ^	1 x 2.52461941829033575705868839676720557403112578229 3,

Example: Computing eigenvalues of Wilkinson matrix of order n using 40 and 60 digits

```
10 ^    -1 x -9.641321726904788469203807653558725272820102494785,
10 ^    -1 x  2.126628425131107295702128494459102204898454357475,
10 ^     0 x  1.078164406734610586751527260933515250175346242597,
10 ^     0 x  1.654730092254134485579766753257886141779285333348,
10 ^     0 x  2.400771751580549653471719453673745125831512658936,
10 ^     0 x  2.613723558709657970251705419366973354981509584181,
10 ^     0 x  3.486274955522127377316131346219042782906283078552,
10 ^     0 x  3.517639311060085206913357746819011821997391706608,
10 ^     0 x  4.498968620932769352208565164016883576924323795219,
10 ^     0 x  4.501192469653747967060030800142379067747029642296,
10 ^     0 x  5.499953844648219428838229822039180544884672380507,
10 ^     0 x  5.500050238625574493614629202431453164635212134466,
10 ^     0 x  6.499998623277988915591995913147693974561916204077,
10 ^     0 x  6.500001455954136799897192333983951377827078801077,
10 ^     0 x  7.499999970478811371657446880312535145009630930559,
10 ^     0 x  7.500000030730059526457051748353478345759432546457,
10 ^     0 x  8.499999999521540927978771807129468065254241367295,
10 ^     0 x  8.500000000493206277760349451609484527161829931395,
10 ^     0 x  9.499999999993923902705172850882166649743950242226,
10 ^     0 x  9.500000000006222657447847663151767192196337560297,
10 ^     1 x  1.049999999999993785719572185114285224895492394635,
10 ^     1 x  1.050000000000006335023029689638865159736459003164,
10 ^     1 x  1.149999999999999476917289260533589466238710695 79,
10 ^     1 x  1.150000000000000005314587972139552209177656679 2268,
10 ^     1 x  1.249999999999999999659868424472185766044029876359,
10 ^     1 x  1.250000000000000004024938536517203064378504554 0,
10 ^     1 x  1.350000000000000004416996709974985823934933254642,
10 ^     1 x  1.350000000000000004421443532537888105779619890527,
10 ^     1 x  1.450000000000000057424636181085647939243013823 4372,
10 ^     1 x  1.450000000000000574246384732497574527508867451697,
10 ^     1 x  1.550000000000006197895720929520873154825160583877 7,
10 ^     1 x  1.550000000000006197895720939799597532956911737456 1,
10 ^     1 x  1.650000000005448810799042914329792294509710737528,
10 ^     1 x  1.650000000005448810799042914734654139270601325 12,
10 ^     1 x  1.750000000380812688353694056838881030953392574759,
10 ^     1 x  1.750000000380812688353694056840291039123740838877,
10 ^     1 x  1.850000020507043780031864988451630954352408854039,
10 ^     1 x  1.850000020507043780031864988451635333251853861565,
10 ^     1 x  1.950000815867294501046405034523704915813165864541,
10 ^     1 x  1.950000815867294501046405034523704928010408453453,
10 ^     1 x  2.050022568018517034412527273640041808200744223750,
10 ^     1 x  2.050022568018517034412527273640041808231370908819,
10 ^     1 x  2.150395200266536132836611456811806949641606804821,
10 ^     1 x  2.150395200266536132836611456811806949641675645570,
10 ^     1 x  2.253894111930644088976740590263711884819586263287,
10 ^     1 x  2.253894111930644088976740590263711884819586491979,
10 ^     1 x  2.371067864733304648832769963393362026697201398322,
10 ^     1 x  2.371067864733304648832769963393362026697201896063,
10 ^     1 x  2.524619418290335757058688396767205574031125611895,
10 ^     1 x  2.524619418290335757058688396767205574031125782293,

10 ^    -1 x -9.641321726904788469203807653558725272820117108072431812438205,
10 ^    -1 x  2.126628425131107295702128494459102204898455201045349204017048,
10 ^     0 x  1.078164406734610586751527260933515250175346019271672542088771,
10 ^     0 x  1.654730092254134485579766753257886141779285140008442589285019,
10 ^     0 x  2.400771751580549653471719453673745125831511857709953513876728,
10 ^     0 x  2.613723558709657970251705419366973354981508439411334331604370,
10 ^     0 x  3.486274955522127377316131346219042782906283490968388275075914,
10 ^     0 x  3.517639311060085206913357746819011821997392015649007013593949,
10 ^     0 x  4.498968620932769352208565164016883576924323049729698939830236,
10 ^     0 x  4.501192469653747967060030800142379067747028959013064519485706,
10 ^     0 x  5.499953844648219428838229822039180544884670563889625419841055,
10 ^     0 x  5.500050238625574493614629202431453164635210303483122503668483,
10 ^     0 x  6.499998623277988915591995913147693974561916330393811371165559,
10 ^     0 x  6.500001455954136799897192333983951377827078908076535885218844,
10 ^     0 x  7.499999970478811371657446880312535145009630113259921077898123,
10 ^     0 x  7.500000030730059526457051748353478345759431690104142798319905,
10 ^     0 x  8.499999999521540927978771807129468065254240939765221646674981,
10 ^     0 x  8.500000000493206277760349451609484527161829486278766873006457,
10 ^     0 x  9.499999999993923902705172850882166649743949997059140087143068,
10 ^     0 x  9.500000000006222657447847663151767192196337321918215996617 56,
10 ^     1 x  1.049999999999993785719572185114285224895493284131284837207 42,
10 ^     1 x  1.050000000000006335023029689638865159736458990204293781960215,
10 ^     1 x  1.149999999999999476917289260533589466238710228812695656924316,
10 ^     1 x  1.150000000000000005314587972139552209177656674820921638372022,
10 ^     1 x  1.249999999999999999659868424472185766044029767935111970283586,
10 ^     1 x  1.250000000000000004024938536517203064378504291392918449961 99,
10 ^     1 x  1.350000000000000004416996709974985823934933350683994156471134,
10 ^     1 x  1.350000000000000004421443532537888105779619989886566224208740,
10 ^     1 x  1.450000000000000057424636181085647939243013839079510067231802 4,
10 ^     1 x  1.450000000000000574246384732497574527508867601049349463260733 1,
10 ^     1 x  1.550000000000006197895720929520873154825160594979070028590499 5,
10 ^     1 x  1.550000000000006197895720939799597532956911748415411324841940 4,
10 ^     1 x  1.650000000005448810799042914329792294509710765503009174611 75,
10 ^     1 x  1.650000000005448810799042914732446541392706015712616612430620 7,
10 ^     1 x  1.750000000380812688353694056838881030953392375964320796572632,
10 ^     1 x  1.750000000380812688353694056840291039123740646782277878662806,
10 ^     1 x  1.850000020507043780031864988451630954352408930096037766099131,
10 ^     1 x  1.850000020507043780031864988451635333251853935993717892930242,
10 ^     1 x  1.950000815867294501046405034523704915813165780014921711917754,
10 ^     1 x  1.950000815867294501046405034523704928010408367449461931580531,
10 ^     1 x  2.050022568018517034412527273640041808200744365950473005838132,
10 ^     1 x  2.050022568018517034412527273640041808231371048706371921123406,
10 ^     1 x  2.150395200266536132836611456811806949641606957197298629818159,
10 ^     1 x  2.150395200266536132836611456811806949641675947765732646192777,
10 ^     1 x  2.253894111930644088976740590263711884819586263247919211944053,
10 ^     1 x  2.253894111930644088976740590263711884819586347193253721016049,
10 ^     1 x  2.371067864733304648832769963393362026697201890611963941496156,
10 ^     1 x  2.371067864733304648832769963393362026697201890706449446616385,
10 ^     1 x  2.524619418290335757058688396767205574031125780246667015958942,
10 ^     1 x  2.524619418290335757058688396767205574031125780246674399808560,
```

Nonlinear Equations

Nonlinear Equations – finding "zeros" of

$$f(x) = 0$$

Common solvers:

- **PETSc** – e.g. Newton-Krylov method.
- **MINPACK**
- **TAO** – Toolkit for Advanced Optimization (Argonne).
- **Harwell** Subroutine Library (**HSL**) – A collection of state-of-the-art Fortran code for large scale scientific computing, developed by NAG and others.
- **IMSL**
- **NAG**

See also in PDEs later.

Examples

Example: Solving nonlinear equation using PETSc. For nonlinear equation

$$F(x) = 0, \quad x \in \mathrm{R}^n.$$

Let $x^{(i)}$ be the approximation and Δx a correction such that $F(x^{(i)} + \Delta x) = 0.$ By Taylor expansion

$$F(x^{(i)} + \Delta x) = 0 = F(x^{(i)}) + F'(x^{(i)})\Delta x + O(\Delta x^2)$$

we have the **Newton** approximation

$$F'(x^{(i)})\Delta x = -F(x^{(i)} \tag{1}$$

$$x^{(i+1)} = x^{(i)} + \Delta x \tag{2}$$

Eq. (1) is mostly solved using **Krylov** subspace method.

When solving PDEs, we often end up with a system of nonlinear equations.

Optimization

Optimization Problems

$$\min_x f(x), \quad \text{s.t. } g(x) = 0, \text{ and, } h(x) \leq 0.$$

Commonly used packages

- **APPSPACK (-> HOPSPACK)** – Asynchronous parallel pattern search with derivative free, by Sandia National Laboratory
- **GSL**
- **NAPACK**
- **MINPACK**
- **OPT++** – An OO nonlinear optimization library (Sandia).
- **TAO** – Toolkit for Advanced Optimization (Argonne).
- **HSL** (Harwell Subroutine Library)
- **IMSL**
- **NAG**

Examples

Example: Data fitting with nonlinear least squares

$$f(x; a, b, \lambda) = a \exp(-\lambda x) + b$$

to fit observation data y_i with uncertainty σ_i. The goal is to find parameters a, λ and b such that

$$\chi(a, b, \lambda)^2 = \sum_{i=1}^{n} \left(\frac{y_i - f(x_i; a, b, \lambda)}{\sigma_i} \right)^2 .$$

Two methods are commonly used, both work well

- Levenberg-Marquardt – requires computing the derivatives.
- Nelder-Mead – simplex search method, does not require to compute derivative information.
- Both methods are available in GSL.

Examples

Example: Using derivatives

```
#include <stdlib.h>
#include <stdio.h>
#include <gsl/gsl_vector.h>
#include <gsl/gsl_blas.h>
#include <gsl/gsl_multifit_nlin.h>
#include "expfit.c"
#define N 40
void print_state (size_t iter, gsl_multifit_fdfsolver * s);

int main (void) {
    const gsl_multifit_fdfsolver_type *T;
    gsl_multifit_fdfsolver *s;
    int status; unsigned int i,
    iter = 0;
    const size_t n = N;
    const size_t p = 3;
    gsl_matrix *covar = gsl_matrix_alloc (p, p);
    double y[N], sigma[N];
    struct data d = { n, y, sigma};
    gsl_multifit_function_fdf f;
    double x_init[3] = { 1.0, 0.0, 0.0 };
    gsl_vector_view x = gsl_vector_view_array (x_init, p);
    const gsl_rng_type * type;
    gsl_rng * r;
    gsl_rng_env_setup();
    type = gsl_rng_default;
    r = gsl_rng_alloc (type);
```

```
f.f = &expb_f;
f.df = &expb_df;
f.fdf = &expb_fdf;
f.n = n;
f.p = p;
f.params = &d;

T = gsl_multifit_fdfsolver_lmsder;
s = gsl_multifit_fdfsolver_alloc (T, n, p);
gsl_multifit_fdfsolver_set (s, &f, &x.vector);
print_state (iter, s);

do {
    iter++;
    status = gsl_multifit_fdfsolver_iterate (s);
    printf ("status = %s\n",
    gsl_strerror (status));
    print_state (iter, s);
    if (status) break;
    status = gsl_multifit_test_delta (s->dx, s->x, 1e-4, 1e-4);
}
while (status == GSL_CONTINUE && iter < 500);
… …
```

Example from GSL

Examples

Example: The use of Nelder-Mead simplex method to search the minimum

```
int main(void)
{
   double par[5] = {1.0, 2.0, 10.0, 20.0, 30.0};
   const gsl_multimin_fminimizer_type *T =
        gsl_multimin_fminimizer_nmsimplex2;
   gsl_multimin_fminimizer *s = NULL;
   gsl_vector *ss, *x;
   gsl_multimin_function minex_func;
   size_t iter = 0;
   int status; double size; /* Starting point */

   x = gsl_vector_alloc (2);
   gsl_vector_set (x, 0, 5.0);
   gsl_vector_set (x, 1, 7.0); /* Set initial step sizes to 1 */
   ss = gsl_vector_alloc (2);
   gsl_vector_set_all (ss, 1.0); /* Initialize method and iterate */

   minex_func.n = 2;
   minex_func.f = my_f;
   minex_func.params = par;
   s = gsl_multimin_fminimizer_alloc (T, 2);
   gsl_multimin_fminimizer_set (s, &minex_func, x, ss);

   do {
      iter++;
      if (status = gsl_multimin_fminimizer_iterate(s)) break;
      size = gsl_multimin_fminimizer_size (s);
      status = gsl_multimin_test_size (size, 1e-2);
      if (status == GSL_SUCCESS) { printf ("converged to minimum
        at\n"); }
      printf ("%5d %10.3e %10.3e f() = %7.3f size = %.3f\n", iter,
        gsl_vector_get (s->x, 0), gsl_vector_get (s->x, 1), s->fval, size);
   }
   while (status == GSL_CONTINUE && iter < 100);
   … …
   return status;
}
```

Example from GSL

Integration

Numerical Integration

$$I = \int \cdots \int_D f(x_1, \ldots, x_n) dx_1 \cdots dx_n, \quad D \subseteq \mathrm{R}^n$$

Commonly use software packages

- **QUADPACK**
- **GSL** – Using QUADPACK (reimplemented) and Monte-Carlo based.
- **Cuba** – A library for multidimensional integrations on $[0, 1]^n$, containing
  - One unified interface to three Monte-Carlo based (Vegas, Suave, Divonne) using different sampling schemes, and one deterministic (Cohre) algorithms;
  - Written in C, has C/C++, Fortran and Mathematica interfaces;
  - Parallelization capable on multicore (shared memory) architecture.

  See T. Hahn, "*Cuba: A library for multidimensional numerical integration*", 2014.
- IMSL
- NAG
- HSL

Matlab

```
I = dblquad(@(x,y) cos(4*x - 3*y + 2*pi), 0,1,0,1)
```

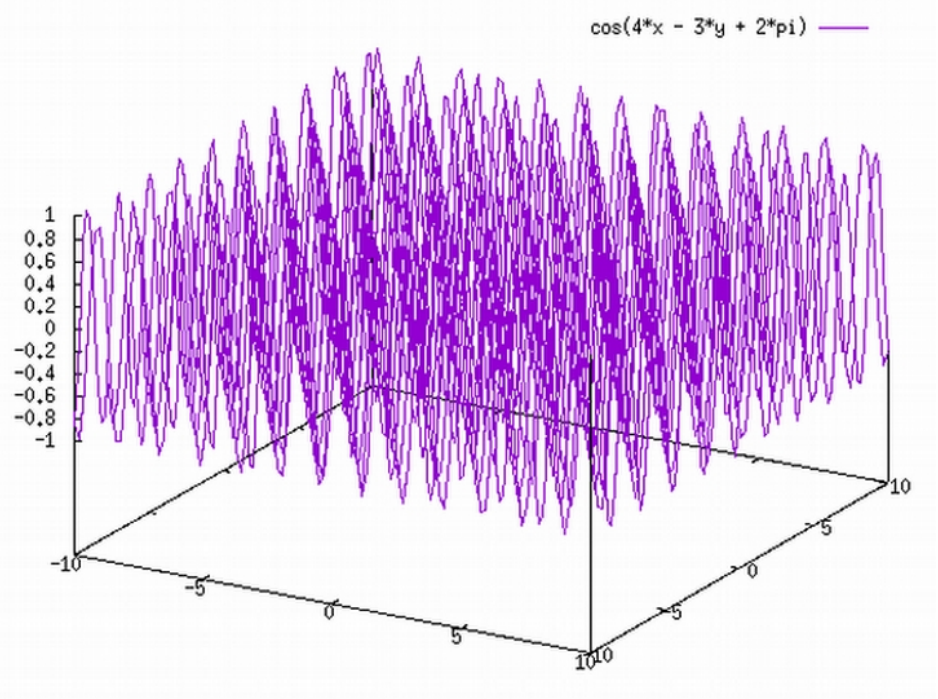
Via Cuba

```
! The calling routine
... ...
call cuba(method, ndim, ncomp, integrand, &
  integral, error, prob)
... ...

! User defined integrand f(x[,...])
integer function integrand(ndim, xx, ncomp, f)
  implicit none
  integer ndim, ncomp
  double precision xx(*), ff(*)
  parameter (pi = 3.14159265358979323846D0)

  x = xx(1)
  y = xx(2)
  f(1) = cos(4.0d0*x - 3.0d0*y + 2.0*pi)
  Integrand = 0
end function integrand
```

Example



$$I = \int_0^1 \int_0^1 \cos(4x - 3y + 2\pi) dx dy$$

ODE Solvers

ODEs

$$\frac{dy}{dt} = f(t, y), \quad y(0) = y_0$$

Commonly used packages

- **odeint (Boost C++ Libraries)**
- **Intel ODE solvers**
- **SUNDIALS** – A suite of solvers, parallel processing ready.
- **ODELPACK**
- **GSL**
- IMSL
- NAG
- HSL

ODE solvers – functional components

- The **function** – to evaluate the right-hand-side f(t,y)
- The "**stepper**" – integration scheme, e.g. explicit, implicit methods
- The **Jacobian** of f(t,y) – when implicit methods are employed, there might be a need to compute the Jacobian at each time step.

ODEs (cont'd)

- **Boost.Numeric.Odeint**
  - C++ interface, simple and clean
  - User to provide **RHS**, and **Jacobian** if necessary
  - Contains a number of integration schemes – ***steppers*** (including **multistep** and **adaptive step control** schemes) – and also allow user to supply a stepper as well
  - Can be used to solve stiff ODEs.

Lorenz system

$$\begin{aligned} \dot{x} &= \sigma(y - x) \\ \dot{y} &= Rx - y - xz \\ \dot{z} &= -bz + xy \end{aligned}$$

Example

```
#include <iostream>
#include <boost/array.hpp>

#include <boost/numeric/odeint.hpp>

using namespace std;
using namespace boost::numeric::odeint;

const double sigma = 10.0;
const double R = 28.0;
const double b = 8.0 / 3.0;

typedef boost::array< double , 3 > state_type;

void lorenz( const state_type &x , state_type &dxdt , double t )
{
    dxdt[0] = sigma * ( x[1] - x[0] );
    dxdt[1] = R * x[0] - x[1] - x[0] * x[2];
    dxdt[2] = -b * x[2] + x[0] * x[1];
}

void write_lorenz( const state_type &x , const double t )
{
    cout << t << '\t' << x[0] << '\t' << x[1] << '\t' << x[2] << endl;
}

int main(int argc, char **argv)
{
    state_type x = { 10.0 , 1.0 , 1.0 }; // initial conditions
    integrate( lorenz , x , 0.0 , 25.0 , 0.1 , write_lorenz );
}
```

ODEs (cont'd)

- **Intel ODE solver** (NOT part of MKL)
  - Fortran, C/C++ interfaces;
  - User to provide **RHS** and **Jacobian**;
  - Contains different ***steppers***;
  - Considered effective for stiff ODEs.

A stiff problem

$$\begin{aligned} \dot{x} &= -0.013x - 1000xz, \\ \dot{y} &= -2500yz, \\ \dot{z} &= -0.013x - 1000xz - 2500yz. \end{aligned}$$

```
void f(int *n, double *t, double *y, double *dydt)
{
   dydt[0] = -0.013*y[0] - 1000.0*y[0]*y[2];
   dydt[1] = -2500.0*y[1]*y[2];
   dydt[2] = -0.013*y[0] - 1000.0*y[0]*y[2] - 2055.0*y[1]*y[2];
}
```

```
void df(int *n, double *t, double *y, double *jac)
{
   jac[0] = -0.013 - 1000.0*y[2];
   jac[1] = 0.0;
   jac[2] = -0.013 - 1000.0*y[2];

   jac[3] = 0.0;
   jac[4] = -2500.0*y[2];
   jac[5] = -2055.0*y[2];

   jac[6] = -1000.0*y[0];
   jac[7] = -2500.0*y[1];
   jac[8] = -1000.0*y[0] - 2055.0*y[1];
}
```

```
#include <stdio.h>
#include "intel_ode.h"

int main(void)
{
   int n, ierr, i;
   int kd[3], ipar[128]; // Set ipar size to 128
   double t_0, t, t_end, h_0, h, hm, ep, tr;
   double y[3], dpar[39]; // dpart size: max{13*n,(7+2*n)*n}=max{39,39}=39

   for (i=0;i<128;i++) ipar[i]=0;

   // Set initial conditions
   read_data(&n, &t_0, &t_end, &h_0, &hm, &ep, &tr, y);

   // Solve ODE using implicit solver, with output of intermediate values
   t = t_0;
   h = h_0;
   printf("\nSolution repeated:\n");
   printf("%g,%g,%g,%g,%g\n",t,h,y[0],y[1],y[3]);

   do
   {
      ipar[2] = 1;    // To use Merson's multistep method
      ipar[3] = 1;    // To exit after each step (for intermediate values)
      ipar[4] = 1;    // To use provided Jacobian function
      dodesol_mk52lfn(
         ipar,&n,&t,&t_end,y,  f,  df,   &h,&hm,&ep,&tr,dpar,kd,&ierr
      );
      printf("%g,%g,%g,%g,%g\n",t,h,y[0],y[1],y[3]);
   }
   while (t < t_end);

   return 0;
}
```

Example

Numerical Libraries

SUNDIALS – A suite of nonlinear differential/algebraic solvers

- **CVODE –** solves initial value problems for ordinary differential equation (ODE) systems.
- **CVODES –** solves ODE systems and includes sensitivity analysis capabilities (forward and adjoint).
- **IDA** – solves initial value problems for differential-algebraic equation (DAE) systems.
- **IDAS** – solves DAE systems and includes sensitivity analysis capabilities (forward and adjoint).
- **KINSOL** – solves nonlinear algebraic systems.

https://computation.llnl.gov/casc/sundials/main.html

PDE Solvers

Numerical Libraries

PDEs – closely related to the solution of linear systems, e.g. discretizing the equation

$$f(x, y, z, u_{xx}, u_{yy}, u_{zz}, u_{xy}, u_{yz}, u_{xz}, u_x, u_y, u_z) = 0$$

results in linear or nonlinear system of equations.

Commonly used packages

- **PETSc** – A popular toolkit, parallel processing ready. Works on nonlinear/linear systems often derived from numerical discretization of PDE or system of PDEs.
- IMSL
- NAG
- Diffpack
- ANSYS
- COMSOL
- OpenFOAM
- … …

Examples

Example: (Cont'd) Suppose we end up with a nonlinear system

$$F(x) = 0, \quad x \in \mathrm{R}^n.$$

Let $x^{(i)}$ denote the approximation of the discrete values of unknown function from the discretization of the PDE on a grid, we are to solve

$$F'(x^{(i)})\Delta x = -F(x^{(i)}), \tag{1}$$

$$x^{(i+1)} = x^{(i)} + \Delta x. \tag{2}$$

The linear system Eq. (1) is solved using **Krylov** subspace method.

We use PETSc to solve the nonlinear system. We need at the minimum the following storage and function evaluations:

- X – solution vector
- R – residual vector, containing F(X)
- J – the Jacobian F'(X)

Optionally PETSc lets users define

- Krylov method
- preconditioner

Numerical Libraries


```
#include "petscsnes.h"

34: int main(int argc,char **argv)
35: {
36:   SNES         snes;    // nonlinear solver context
37:   KSP          ksp;     // linear solver context
38:   PC           pc;      // preconditioner context
39:   Vec          x,r;     // solution, residual vectors
40:   Mat          J;       // Jacobian matrix
42:   PetscInt     its;
43:   PetscMPIInt  size,rank;
44:   PetscScalar  pfive = .5,*xx;
45:   PetscTruth   flg;
46:   AppCtx       user;    // user-defined work context
47:   IS           isglobal,islocal;

49:   PetscInitialize(&argc,&argv,(char *)0,help);
50:   MPI_Comm_size(PETSC_COMM_WORLD,&size);
51:   MPI_Comm_rank(PETSC_COMM_WORLD,&rank);

53:   // Create nonlinear solver context
56:   SNESCreate(PETSC_COMM_WORLD,&snes);

58:   // Create solutions objects
64:   VecCreate(PETSC_COMM_WORLD,&x);
65:   VecSetSizes(x,PETSC_DECIDE,2);
67:   VecDuplicate(x,&r);

69:   if (size > 1){
        Setup for distributed processing
79:   }

82:   // Create Jacobian matrix data structure
84:   MatCreate(PETSC_COMM_WORLD,&J);
85:   MatSetSizes(J,PETSC_DECIDE,PETSC_DECIDE,2,2);
86:   MatSetFromOptions(J);

88:   PetscOptionsHasName(PETSC_NULL,"-hard",&flg);
89:   if (!flg) {
93:     SNESSetFunction(snes,r,FormFunction1,&user);
98:     SNESSetJacobian(snes,J,J,FormJacobian1,PETSC_NULL);
99:   } else {
100:    if (size != 1)
          SETERRQ(1,"This case is a uniprocessor example only!");
101:    SNESSetFunction(snes,r,FormFunction2,PETSC_NULL);
102:    SNESSetJacobian(snes,J,J,FormJacobian2,PETSC_NULL);
103:  }

106:  // Customize nonlinear solver; set runtime options
113:  SNESGetKSP(snes,&ksp);
114:  KSPGetPC(ksp,&pc);
115:  PCSetType(pc,PCNONE);
116:  KSPSetTolerances(ksp,1.e-4,\
        PETSC_DEFAULT,PETSC_DEFAULT,20);

118:  // Set SNES/KSP/KSP/PC runtime options
      SNESSetFromOptions(snes);
```

Example

Numerical Libraries


```
128:  //   Evaluate initial guess; then solve nonlinear system
130:  if (!flg) {
131:    VecSet(x,pfive);
132:  } else {
133:    VecGetArray(x,&xx);
134:    xx[0] = 2.0; xx[1] = 3.0;
135:    VecRestoreArray(x,&xx);
136:  }

144:  SNESSolve(snes,PETSC_NULL,x);
145:  SNESGetIterationNumber(snes,&its);
146:  if (flg) {
147:    Vec f;
148:    VecView(x,PETSC_VIEWER_STDOUT_WORLD);
149:    SNESGetFunction(snes,&f,0,0);
150:    VecView(r,PETSC_VIEWER_STDOUT_WORLD);
151:  }

153:  PetscPrintf(PETSC_COMM_WORLD,"number of Newton
      iterations = %D\n\n",its);

156:  //   Free work space
160:  VecDestroy(x); VecDestroy(r);
161:  MatDestroy(J); SNESDestroy(snes);
162:  if (size > 1){
163:    VecDestroy(user.xloc);
164:    VecDestroy(user.rloc);
165:    VecScatterDestroy(user.scatter);
166:  }
167:  PetscFinalize();
168:  return 0;
169: }
```

Example

Summary

- PETSc' uses the following key opaque objects:
  - **SNES** - Solver object;
  - **KSP** - KSP object;
  - **PC** - Preconditioning object;
- PETSc also defines opaque matrix and vector types. Need interface routines to set and get values.
- User to define the computational grid.
- User to define routines for evaluating the function F(x) and Jacobian F'(x) as ***CALLBACK*** functions installed in SNES object.
- The solving procedure is self contained in the call to **SNESSolve()**.
- Built with parallel processing enabled.

FFT

Numerical Libraries

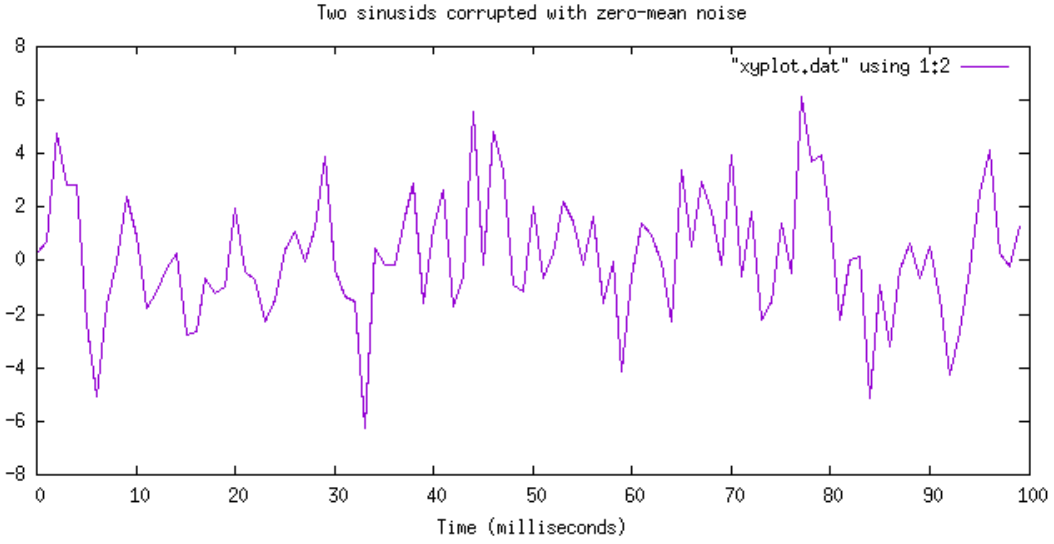
FFT – The best choice is ***FFTW***

- C/C++, Fortran interfaces;
- Used by Matlab, Octave, R, etc;
- For both serial and parallel processing – threaded and MPI.

Example – 1d signal processing via ***FFTW***

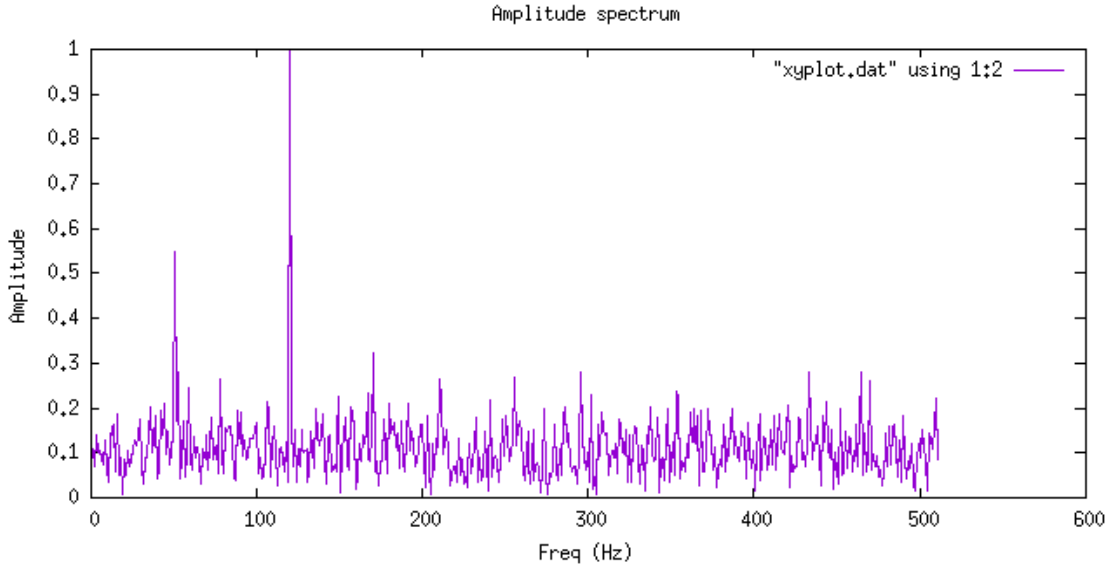
Input

$$x(t) = 0.7\sin(2\pi 50t) + \sin(2\pi 120t) + r$$



Output

$$X(\omega) = FFT(x(t))$$



Matlab

```
Fs = 1000;                % Sampling frequency
T = 1/Fs;                 % Sample time
L = 1000;                 % Length of signal
t = (0:L-1)*T;            % Time vector

% Create signal vector: Two sinusoid corrupted by noise
x = 0.7*sin(2*pi*50*t) + sin(2*pi*120*t) + 2*randn(size(t));
figure(1)
plot(Fs*t(1:100),x(1:100))
title('Signal Corrupted with Zero-Mean Random Noise')
xlabel('time (milliseconds)')
input('Press ENTER key to continue');

% Performa FFT on vector x, output is y
NFFT = 2^nextpow2(L); % Next power of 2 from length of x
y = fft(x,NFFT)/L;

% Plot single-sided amplitude spectrum.
f = Fs/2*linspace(0,1,NFFT/2+1);
figure(2)
plot(f,2*abs(y(1:NFFT/2+1)))
title('Single-Sided Amplitude Spectrum of x(t)')
xlabel('Frequency (Hz)')
ylabel('|X(f)|')
input('Press ENTER key to continue');
```

Fortran

Example

```
program fftw_signal
  use fftw
  use graphics
  implicit none
  integer, parameter :: dp=kind(1.0d0)
  real, parameter :: PI=3.14159265358979323846
  integer :: k = 1024       ! Sampling frequency
  real(dp) ::time           ! Sample time
  integer :: i, n = 1024    ! Length of signal
  real(dp) :: t(1024), f(1024), x(1024), noise(1024)
  complex(dp) :: y(1024)
  integer(8) :: plan

  ! Create time vector
  time = 1.0/k
  t = (/(i,i=0,n-1)/)*time

  ! Create signal vector: Two sinusoids corrupted by noise
  call randn(noise)
  x = 0.7*sin(2*PI*50*t) + sin(2*PI*120*t) + 2*noise
  call plot(k*t(1:100),x(1:100))

  ! Perform FFT on vector x, output is y
  call dfftw_plan_dft_r2c_1d(plan, n, x, y, FFTW_ESTIMATE)
  call dfftw_execute_dft_r2c(plan, x, y)
  call dfftw_destroy_plan(plan)
  y = y/k

  ! Create frequence vector
  f = 0.5*k*(/(i,i=0,n-1)/)/(n/2+1)
  call plot(f,2*abs(y(1:n/2+1)),ymax=1.0d0,xlabel='Freq',ylabel='Amplitude')
end program fftw_signal
```

FFTW functions

- fftw_plan ***x*fftw_plan_*type*_*dim*** (fftw_plan **plan**, int n, fftw_*dtype* ***in**, fftw_*dtype* ***out**, unsigned flag)
- void ***x*fftw_execute_*type*** (fftw_plan **plan**, in, out)
- void ***x*fftw_destroy_plan** (fftw_plan **plan**)

Example

- dfftw_plan_dft_r2c_1d(plan, n, x, y, FFTW_ESTIMATE)
- dfftw_execute_dft_r2c(plan, x, y)
- dfftw_destroy(plan)
- … …
- dfftw_execute_dft_c2r(plan, x, y)

FFTW_MEASURE
FFTW_PATIENT
FFTW_EXHAUSTIVE
FFTW_WIDSOM_ONLY
... ...

Random Number Generators

Random Number Generators

- **GSL** – has a collection of high quality RNGs:
  - 60+ RNGs;
  - 40+ distributions;
  - Easy to use, callable from C/C++ and Fortran.
- **SPRNG** – Scalable parallel random number generators library.
  - Work with MPI
  - Currently has uniform distribution only.
- **RANLIB** – Fortran/C libraries from netlib.
- HSL
- IMSL
- NAG

Example: Random number generation using GSL

- GSL includes a set of high quality RNGs, set by the call to **gsl_rng_alloc**(*type*)
- The usage of GSL RNGs is relatively simple, involving the following

```
#include <gsl/gsl_rng.h>                  // GSL RNG headers
#include <gsl/gsl_randist.h>              // Different random number distributions

rng_h = gsl_rng_alloc(gsl_rng_mt19937); // E.g. to use the MT-19937algorithm, default

gsl_rng_set(rng_h,seed);                  // Set seed

r = gsl_rng_uniform(rng_h);               // Call to get the next random number
r = gsl_rng_uniform_pos(rng_h);           // Return a positive number
r = gsl_rng_uniform_int(rng_h, N);        // Return an integer between 0 and N-1

or

r = gsl_ran_poisson(rng_h, λ);            // Get the next random number
r = gsl_ran_gaussian(rng_h, σ);           // Get the next from Gaussian distribution with var=1.0
```

Examples

Example: Generating random numbers from Poisson distribution using GSL

```
#include <stdio.h>
#include <time.h>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>

const gsl_rng *rng_h;      /* global rand number generator */

int main (int argc, char *argv[]) {
   unsigned long seed;
   int i, k, n = 10;
   double lambda = 3.0;

   get the  lambda from command line

   rng_h = gsl_rng_alloc(gsl_rng_mt19937);
   srand(time(NULL));              /* initialization for rand() */
   seed = rand();                  /* returns a non-negative integer */
   gsl_rng_set (rng_h, seed);      /* seed the RNG */

   for (i = 0; i < n; i++) {
      k = gsl_ran_poisson (rng_h, lambda);
      printf (" %u", k);
   }

   gsl_rng_free(rng_h);
   return 0;
}
```

Example: An Fortran implementation of randn() using GSL

```
subroutine randn(r)
  implicit none
  integer, parameter :: dp=kind(1.0d0)
  real(dp) :: r(:)
  integer :: n

  n = size(r)
  call crandn (n, r)
end subroutine randn
```

Fortran interface

```
#include <stdio.h>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>

double crandn_ (int *n, double *ra)
{
   int i;
   const gsl_rng_type * T;
   gsl_rng * r;

   gsl_rng_env_setup();
   T = gsl_rng_default;
   r = gsl_rng_alloc (T);

   for (i = 0; i < *n; i++)
      ra[i] = gsl_ran_gaussian(r, 1.0);
   gsl_rng_free (r);
}
```

C code

Examples

SPRNG: Some properties:

- Supports both serial and parallel programs.
- Several RNGs are employed. For instance, 48-bit linear congruential generator

  $x(n) = a*x(n-1) + p \% M$

  where a is a constant, p is a prime number and $M = 2^{48}$.
- It lets users to choose what to use via the call to

  init_sprng(seed,SPRNG_DEFAULT,***rng_type***)
- It has all the serial RNGs' properties.
- It can generate multiple ***streams*** in parallel that are not correlated.
- Same "***seed***" – encoding of *starting state*, can generate distinct streams. This is convenient for users!
- Slightly different interfaces: serial vs. MPI and simple vs. default.
- Checkpoint capable.
- Unlike GSL, it does not come with functions for variety of distributions.

Examples

C++

```
#include <iostream>
#include <mpi.h>
#define SIMPLE_SPRNG
#define USE_MPI // use MPI to find number of processes
#include "sprng.h"
#define SEED 985456376
using namespace std;

int main(int argc, char *argv[]) {
   double rn; int i, myid; int gtype;

   MPI::Init(argc, argv); /* Initialize MPI */
   myid = MPI::COMM_WORLD.Get_rank();

   if(myid == 0) {
#include "gen_types_menu.h“
      set rng;
   }
   MPI::COMM_WORLD.Bcast(&gtype,1,MPI::INT,0)
   init_sprng(SEED,SPRNG_DEFAULT,gtype);

   for (i=0;i<10000;i++) {
      rn = sprng();
      perform calculations
   }

   MPI::Finalize();
   return 0;
}
```

Fortran

```
program sprngf_mpi
  implicit none
#define SIMPLE_SPRNG ! simple interface
#define USE_MPI ! use MPI to find number of processes
  use mpi
#include "sprng_f.h"
  SPRNG_POINTER:: junkPtr
  real(8):: rn
  integer:: seed, i, myid, ierr, junk, rng

  call MPI_INIT(ierr)
  call MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
  seed = 985456376

  if (myid .eq. 0) then
#include "genf_types_menu.h“
    set rng
  endif
  call MPI_BCAST(rng, 1, MPI_INTEGER,0,MPI_COMM_WORLD,ierr)
  junkPtr = init_sprng(seed,SPRNG_DEFAULT,rng)

  do i = 1, 10000
    rn = sprng()
    perform calculations
  enddo
  call MPI_FINALIZE(ierr)
end program sprngf_mpi
```

Multipurpose Libraries

Numerical Libraries

AMD Core Math Library (ACML)

- BLAS
- LAPACK
- FFT
- Random number generators

Fortran, C APIs

Numerical Libraries

Intel Math Kernel Library (MKL)

- BLAS
- Sparse BLAS
- LAPACK
- ScaLAPACK
- Extended eigensolvers
- Direct and **Iterative Sparse Solver routines**
- **Vector Mathematical Library (VML) functions**
- **Vector Statistical Library (VSL) functions**
- FFT
- PDE solvers.
- **Optimization** solver routines for solving nonlinear least squares problems through the Trust-Region (TR) algorithms and computing Jacobi matrix by central differences
- **Data fitting**, differentiation and integration, and search
- GMP arithmetic functions

Fortran, C APIs

Numerical Libraries

International Mathematical Software Library (IMSL)

- ***Mathematics***
  - Linear algebra
  - Eigensystems
  - **Interpolation and approximation**
  - **Integration and differentiation**
  - Differential equations
  - Transforms
  - **Optimizations**
  - Nonlinear equations
  - Vector, matrix operations
  - Basic linear algebraic operations
  - Utilities
  - Parallel processing on distributed systems
- ***Statistics***
  - Basic stats
  - Regression
  - Correlations
  - Analysis of variance
  - Categorical and discrete data analysis
  - Nonparametric stats
  - Test of goodness of fit and randomness
  - Time series analysis and forcasting
  - Covariance structure and factor analysis
  - Discriminant analysis
  - Cluster analysis
  - Sampling
  - Survival analysis
  - Multidimensional scaling
  - Density and hazard estimation
  - Line printer graphics
  - Probability distribution functions and inverses
  - Random number generation
- ***Special Functions***

Fortran, C/C++ and other language APIs

Numerical Algorithms Group (NAG) Library

Numerical Libraries

- **Optimization**
- Linear, quadratic, integer and nonlinear programming and least squares problems
- ODEs, PDEs and mesh generation
- Solution of dense, banded and sparse linear equations and eigenvalue problems
- Linear and nonlinear least squares problems
- Curve and surface fitting and interpolation
- **Special functions**
- Numerical integration and integral equations
- Nonlinear equations
- **Option pricing formulae**
- **Wavelet Transforms**

Statistics Libraries

- Random number generation
- Simple calculations on statistical data
- Correlation and regression analysis
- Multivariate methods
- Analysis of variance and contingency table analysis
- Time series analysis
- Nonparametric statistics

Fortran, C APIs

GNU Scientific Library (GSL) – A collection of bit of everything…

- **Complex Numbers**
- Special Functions
- Permutations
- **Random Numbers**
- Quasi-Random Sequences
- Random Distributions
- Statistics
- Monte Carlo
- Simulated Annealing
- Series Acceleration
- Minimization
- Least-Squares Fitting
- Physical Constants
- Discrete Wavelet Transforms
- Roots of Polynomials
- **Vectors and Matrices**
- Sorting
- Linear Algebra
- Fast Fourier Transforms
- Transforms
- Interpolation and Approximations
- Numerical Differentiation
- Quadratures and Integration
- Differential Equations
- Root-Finding

C/C++ only!

Numerical Libraries

Resources at *netlib* – A collection of packages

- **Key packages**
  - LINPACK
  - EISPACK
  - BLAS
  - LAPACK
  - ScaLAPACK
  - MINPACK
  - NAPACK
  - MPFUN
- **Many other packages**

http://www.netlib.org/

Numerical Libraries

PETSc – Portable, Extensible Toolkit for Scientific Computing

- Vector, matrix objects and operations
- Finite difference approximation of Jacobians
- Linear system (dense and sparse) solvers: **LU** and Krylov subspace method with preconditioners (**KSP**).
- Nonlinear equation solvers (**SNES**).
- Time stepping (TS) solvers for ODEs.
- Eigenvalue problems with **SLEPc**.

http://www.mcs.anl.gov/petsc/

C/C++ and Fortran interfaces
Evident traces of C/C++ thinking
Works for multiprocessor environment, transparent to users
Error messages hard to comprehend

PETSc 3.1 – A list of packages needed…

Setting up libbtf1.1.0 (1:3.4.0-2ubuntu2) ...
Setting up **libcsparse**2.2.3 (1:3.4.0-2ubuntu2) ...
Setting up libklu1.1.0 (1:3.4.0-2ubuntu2) ...
Setting up libldl2.0.1 (1:3.4.0-2ubuntu2) ...
Setting up **libsuitesparse**-dev (1:3.4.0-2ubuntu2) ...
Setting up **libsuperlu**3 (3.0+20070106-3) ...
Setting up libsuperlu3-dev (3.0+20070106-3) ...
Setting up mpi-default-bin (0.6) ...
Setting up libblacs-mpi1 (1.1-28.2build1) ...
Setting up mpi-default-dev (0.6) ...
Setting up libblacs-mpi-dev (1.1-28.2build1) ...
Setting up libhdf5-openmpi-dev (1.8.4-patch1-2ubuntu3) ...
Setting up libhdf5-mpi-dev (1.8.4-patch1-2ubuntu3) ...

Setting up **libhypre**-2.4.0 (2.4.0b-7) ...
Setting up libhypre-dev (2.4.0b-7) ...
Setting up **libscalapack**-mpi1 (1.8.0-6) ...
Setting up **libmumps**-4.9.2 (4.9.2.dfsg-6) ...
Setting up libscalapack-mpi-dev (1.8.0-6) ...
Setting up libmumps-dev (4.9.2.dfsg-6) ...
Setting up **libscotch**-5.1 (5.1.11.dfsg-7) ...
Setting up libspooles2.2 (2.2-8) ...
Setting up libpetsc3.1 (3.1.dfsg-10ubuntu1) ...
Setting up libspooles-dev (2.2-8) ...
Setting up libscotch-dev (5.1.11.dfsg-7) ...
Setting up libpetsc3.1-dev (3.1.dfsg-10ubuntu1) ...
update-alternatives: using /usr/lib/petscdir/3.1 to provide /usr/lib/petsc (petsc) in auto mode.
update-alternatives: using /usr/lib/petscdir/3.1/aclocal/math-blaslapack.m4 to provide /usr/share/aclocal/math-blaslapack.m4 (math-blaslapack.m4) in auto mode.
Setting up petsc-dev (3.1.dfsg-10ubuntu1) ...

Numerical Libraries

Hypre – A library for solving large, sparse linear systems

- Common **Krylov** subspace based methods.
- Contains a suite of **preconditioners**.
- Uses MPI.

http://acts.nersc.gov/hypre/

Further Readings

- Netlib, http://www.netlib.org/.
- M. T. Heath, "*Scientific Computing: An introductory survey*", 2nd Ed, 2002.
- Victor Eijikhout, "Overview of Iterative Linear System Solver Packages", 1998.
- BLAS Quick References.
- LAPACK Quick References.
- LAPACK User's Guide.
- ScaLAPACK User's Guide.
- PETSc, http://www.mcs.anl.gov/petsc/. In active development, referencing to classic and recent literatures.
- GNU Scientific Libraries, http://www.gnu.org/manual/manual.html.
- Boost C++ libraries, http://www.boost.org/doc/libs/1_57_0/.
- Hardware vendor's sites: AMD, Intel, HP, IBM, Sun, nVIDIA, etc.
- Software vendor's sites: Intel, IMSL (Rogue Waves), NAG, etc.

Q&A