proj2

October 6, 2016

1 Project 2 – Linear and Nonlinear Solvers

1.1 MATH 3316 High Performance Scientific Computing, Fall 2016

Author: Paul Herz

The designated programming language for this project was C++, built in Xcode and compiled with Clang (Apple LLVM 8.0.0) using the C++14 dialect.

1.1.1 Structure of this Project

This project's file structure follows the GNU-style C++ project standard.

```
In [1]: %ls ..
```

```
HPSCProject2.xcodeproj/ data/                   reports/
Makefile                lib/                    src/
bin/                    notebooks/
```

Excluding the `.xcodeproj` file, which was used for integration with the Xcode IDE, debugging, and profiling purposes, each item in the project directory serves the following purpose:

- `Makefile`: GNU Make project build automation definitions
- `bin/`: compiled binaries. Make will put binaries here by default.
- `data/`: where calculated data is stored after program execution. Files are `.txt` files containing real numbers, space delimited to denote row items, and newline delimited to denote new rows. Some data files, whose names end in `_out`, are just piped output from program execution and follow no specific format.
- `lib/`: reused libraries that are not part of this project specifically. Contains my rewrite of the Matrix library (phrz/matrix).
- `notebooks/`: Python 3 Jupyter notebooks, notably the one used to generate this report.
- `reports/`: PDFs generated from the Jupyter notebooks via `nbconvert` and `pdflatex`.
- `src/`: contains the main mathematical routines described in this report, which are used to generate the data and redirected output in `data/`.

1.1.2 Using this Project

Prerequisites

- A Unix or Unix-like OS (e.g. macOS or Linux)
- A compiler with support for C++14 (LLVM or GNU toolchain)
- Python ≥3.5
- The latest Jupyter distribution
- LaTeX with `pdflatex`
- GNU Make ≥3.81

Building this project This project provides several GNU Make targets, with a handful of them being especially useful.

`make all` (default) - will compile binaries, execute them to generate data files, execute Jupyter notebooks under fresh kernels with just-generated data files, and convert them to PDFs in `/reports`.

`make all_bin` - will compile binaries for parts A, B, and C (`vandermonde`, `newton`, and `kepler`) of this project.

`make all_data` - will compile binaries and execute them to generate data files.

`make clean` - will delete all compiled binaries, generated data, executed notebook copies (but not the original notebooks), and **report PDFs**.

Below, find attached the full Makefile:

```
In [2]: %cat ../Makefile
```

```
#
#  Makefile
#  HPSCProject2
#
#  Created by Paul Herz on 10/4/16.
#  Copyright © 2016 Paul Herz. All rights reserved.
#

TARGETA = vandermonde
TARGETB = test_newton
TARGETC = kepler

CXX = g++
CFLAGS = -std=c++14

SRC = src/
LIB = lib/
BIN = bin/
ROOT = $(shell pwd)/
DATA = data/
NB = notebooks/
RP = reports/

AFILES = vandermonde.cpp
```

```
BFILES = test_newton.cpp
CFILES = kepler.cpp
LIBFILES = Vector.cpp Matrix.cpp

NOTEBOOK = $(NB)proj2.ipynb
REPORT = $(RP)proj2.pdf



################################
# All target                   #
################################

all: all_bin all_data all_reports



################################
# Application binaries         #
################################

all_bin: $(TARGETA) $(TARGETB) $(TARGETC)

$(TARGETA):
        $(CXX) $(CFLAGS) -o $(BIN)$(TARGETA) -I $(LIB) $(addprefix $(LIB), $(LIBFI

$(TARGETB):
        $(CXX) $(CFLAGS) -o $(BIN)$(TARGETB) -I $(LIB) $(addprefix $(LIB), $(LIBFI

$(TARGETC):
        $(CXX) $(CFLAGS) -o $(BIN)$(TARGETC) -I $(LIB) $(addprefix $(LIB), $(LIBFI



################################
# Data files                   #
################################

all_data: data_a data_b data_c

data_a: $(TARGETA)
        cd $(BIN); ./$(TARGETA) > $(ROOT)$(DATA)a_out.txt

data_b: $(TARGETB)
        cd $(BIN); ./$(TARGETB) > $(ROOT)$(DATA)b_out.txt

data_c: $(TARGETC)
        cd $(BIN); ./$(TARGETC)
```

```
################################
# Reports                      #
################################

all_reports: $(REPORT)

$(REPORT): all_data $(NOTEBOOK)
        jupyter nbconvert --to pdf --execute $(NOTEBOOK) --output-dir $(ROOT)$(RP)



################################
# Miscellaneous                #
################################

clean:
        rm -f ./$(DATA)/*
        rm -f ./$(BIN)$(TARGETA)
        rm -f ./$(BIN)$(TARGETB)
        rm -f ./$(BIN)$(TARGETC)
        rm -f ./$(NB)*.nbconvert.ipynb
```

1.2 Part A – Vandermonde Matrices

Part A of this project concerned itself with the definition of Vandermonde matrices, implementing them programmatically (namely, their generation given certain parameters), and their deleterious nature in solving linear systems with Gaussian elimination.

1.2.1 Goals

I needed to begin this project with an algorithm to generate Vandermonde matrices from an initial column vector of size n. When my program was asked to generated an n^{th} degree Vandermonde matrix, it needed to first generate a column vector $\mathbf{v}$ consisting of a linear span from $0 \dots 1$ from $v_0 \dots v_{n-1}$. This would be used as the columns of the Vandermonde matrix pre-exponentiation. I would then generate the vector $\mathbf{x}$ of degree n with a uniformly random distribution.

1.2.2 Math Background

Vandermonde Matrix A Vandermonde matrix is a pathologically exemplary ill-conditioned matrix defined as a degree n matrix whose columns are based upon a single n-sized column vector $\mathbf{v}$, such that for each element of the matrix a_{ij}, $a_{ij} = {v_i}^{j-1}$ (assuming one-indexing)

Error Vector The error vector is the absolute value of the elementwise difference of the true solution $\mathbf{x}$ and the Gaussian solution $\hat{\mathbf{x}}$, or $\vec{\epsilon} = |\mathbf{x} - \hat{\mathbf{x}}|$. The *error* is a scalar defined as the two-norm of this error vector.

Residual Vectors For a linear system $\mathbf{Ax} = \mathbf{b}$, the solution found through Gaussian elimination $\hat{\mathbf{x}}$ has inherent error relative to the "true" $\mathbf{x}$, which can be represented through the absolute error function $|\mathbf{A}\hat{\mathbf{x}} - \mathbf{Ax}|$, which through substitution is $|\mathbf{A}\hat{\mathbf{x}} - \mathbf{b}|$. The *residual* is a scalar defined as the two-norm of this residual vector.

Two Norm For a column vector $\mathbf{x}$, of size n and containing elements $x_0 \dots x_{n-1}$, the two norm is the result of the expression:

$$\left(\sum_{0}^{n-1} {x_i}^2 \right)^{1/2}$$

1.2.3 Implementation Requirements

Vandermonde Methods and the Matrix Library I was required to create a file called `vandermonde.cpp` whose main method would iterate the array $5, 9, 17, 33, 65$, and for each such value, create a Vandermonde matrix $\mathbf{A}$ of that degree, and with a randomly-generated $\mathbf{x}$ vector, attempt to solve it. Following that, I was to perform quantitative error analysis by means of the error and residual scalars for the linear system. The purpose of this exercise was to reveal the ill-conditioned nature of the Vandermonde matrix, and the behavior of linear systems involving such a matrix over increasing degree n. The matrix library functions to be used for (1) the linear span of $\mathbf{v}$, the source material for the Vandermonde matrices, (2) the uniform random distribution of $\mathbf{x}$, and (3) the solution of the resultant linear system are `Matrix::linSpace`, `Matrix::random`, and `Matrix::linearSolve`. Note that I am using a custom-rolled variant of the matrix library, of my own design — API calls will vary from the anticipated names.

1.2.4 Implementation

Vandermonde Methods and the Matrix Library For the sake of simplicity, legibility, and conciseness, I wrote the main routine of this program as a loop over the given array of n values. Within the loop, I generated `Vector v` as a linear span, `Vector x` as a random vector, and `Matrix A` as the result of my `makeVandermonde` method, which clearly delineates the concerns of calculation in this exercise. `makeVandermonde` simply uses my `Matrix` class's `mapElements` function to render the aforementioned mapping

$$a_{ij} = {v_i}^{j-1}$$

Although with zero-indexing, as it is performed in the actual code, the formula is

$$a_{rc} = {v_r}^{c}$$

```
In [3]: %cat ../src/vandermonde.cpp
```

```
//
//  vandermonde.cpp
//  HPSCProject2
//
```

```
//  Created by Paul Herz on 9/21/16.
//  Copyright © 2016 Paul Herz. All rights reserved.
//

#include "Matrix.h"
#include "Vector.h"
#include "println.cpp"

using namespace PH;

Matrix makeVandermonde(Vector v) {
        // The Vandermonde matrix of order n is an (n,n) matrix is defined as
        // A(r,c) = v[r] ^ c (assuming zero indexing!)

        Matrix vandermonde = Matrix(v.size(), v.size());

        vandermonde.mapElements([&](double& element, Index r, Index c) {
                element = std::pow(v[r], c);
        });

        return vandermonde;
}

int main(int argc, const char * argv[]) {

        // Solve a linear system Ax=b with a Vandermonde A of order n
        // for each n = 5, 9, 17, 33, 65.
        for(auto n : {5, 9, 17, 33, 65}) {

                println("n =",n,":");

                // To generate the matrix, use vector v with an n-size linear span
                // 0...1 (inclusive), and vector x with n random entries.

                Vector v = Vector::linSpace(0, 1, n);
                Vector x = Vector::random(n);

                Matrix A = makeVandermonde(v);

                // Set b to be equal to A*x so the exact solution is known.

                Vector b = A*x;

                // find x-hat using the linear solver with A and b.

                try {
                        // Copy A and b as they will be modified but we need
                        // the originals for error calculation
```

```
                        auto bTemp = Vector(b);
                        auto ATemp = Matrix(A);
                        Vector xHat = Matrix::linearSolve(ATemp, bTemp);

                        // print the 2-norm of the error vector and the residual ve
                        Vector errorVector = Vector(n);
                        errorVector.mapElements([&](double& element, Index i) {
                                element = std::abs( xHat[i] - x[i] );
                        });

                        Vector residualVector = b - (A * xHat);

                        println("  • Error vector 2-norm:", Vector::norm(errorVecto
                        println("  • Residual vector 2-norm:", Vector::norm(residua

                } catch(std::runtime_error* e) {
                        println("  • Matrix is singular");
                }

                println("--------------------------------");

        }

        // output can be piped to file with "binaryfile > outputfile"

        return 0;
}
```

After generating the n^{th} degree Vandermonde matrix $\mathbf{A}$, all that's left is to perform Gaussian elimination to solve the system $\mathbf{Ax} = \mathbf{b}$ to calculate the approximate value $\hat{\mathbf{x}}$. Note that I produce copies of `A` and `b` — this is due to a quirk of the `Matrix` library remnant of its original design: although not apparent, `A` and `b` are modified in place when passed to `Matrix::linearSolve`. After that, error calculation is performed by generating the error vector as the elementwise operation $\vec{\epsilon} = |\hat{\mathbf{x}} - \mathbf{x}|$, essentially producing a vector of the elementwise relative error of $\hat{\mathbf{x}}$ in approximating $\mathbf{x}$. I display the two-norm of this error vector (the error scalar) using the inbuilt `Vector::norm` method (note that my `Matrix` library includes a distinct `Vector` class, to more closely approximate the familiar behavior of MATLAB). I also calculate the residual vector as the elementwise relative error of $\hat{\mathbf{b}} = \mathbf{A}\hat{\mathbf{x}}$ in approximating $\mathbf{b}$. Similarly, a residual scalar is generated and displayed. No file manipulation for persistence is performed here, but execution output was piped to a file for demonstrative purposes:

Output

```
In [4]: %cat ../data/a_out.txt

n = 5 :
  • Error vector 2-norm: 5.42561
```

```
  • Residual vector 2-norm: 56.5159
------------------------------
n = 9 :
  • Error vector 2-norm: 17.2352
  • Residual vector 2-norm: 419.945
------------------------------
n = 17 :
  • Error vector 2-norm: 36.5481
  • Residual vector 2-norm: 2448.83
------------------------------
n = 33 :
  • Error vector 2-norm: 82.3969
  • Residual vector 2-norm: 15266.1
------------------------------
n = 65 :
  • Error vector 2-norm: 192.933
  • Residual vector 2-norm: 100037
------------------------------
```

1.2.5 Analysis

Due to the exemplary ill-conditioned nature of the Vandermonde matrix, it isn't surprising to find that the error of the result $\hat{\mathbf{x}}$, or the error of the product approximation $\hat{\mathbf{b}}$ (residual) demonstrate runaway behavior as the degree of the matrix increases — to such extent that the results Gaussian elimination become practically useless for most larger matrices of this variety. An ill-conditioned matrix is formally defined as a matrix having a high condition number, which is a calculated value quantifying the "instability" of a linear system, i.e. how susceptible a system is to significant changes in solution given slight modifications (perturbations). Notice how as n increases, the error scalar seems to increase in a more linear fashion, and thus the quality of the approximation of $\mathbf{x}$ decreases in a linear fashion. But then notice how rapidly the residual scalar demonstrates a much more "runaway" effect — it should be noted that in an ill-conditioned system, the residual **cannot** be depended upon as a quantitative metric of system solution accuracy — so despite the outrageous numbers, this is only a side effect of the ill-conditioned system.

1.3 Part A – Newton's Method

Project 2 Part B focused on root finding with Newton's method, and the behavior of the algorithm given certain initial guesses and tolerances. Newton's method is not globally convergent for certain methods, and very often a bad guess will lead to runaway behavior.

1.3.1 Goals

I needed to begin this project with an algorithm to generate Vandermonde matrices from an initial column vector of size n. When my program was asked to generated an n^{th} degree Vandermonde matrix, it needed to first generate a column vector $\mathbf{v}$ consisting of a linear span from $0 \ldots 1$ from $v_0 \ldots v_{n-1}$. This would be used as the columns of the Vandermonde matrix pre-exponentiation. I would then generate the vector $\mathbf{x}$ of degree n with a uniformly random distribution.

1.3.2 Math Background

Newton's Method Newton's method is an iterative algorithm for generating x_{n+1} in terms of x_n. The algorithm is "finished" once it obtains an acceptable approximation of a root for the function $f(x)$. However, in a practical setting, a maximum limit is usually set on iterations in case the method does not converge. The method step is defined as:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

1.3.3 Implementation Requirements

Newton's Method I was provided detailed requirements for my Newton's method implementation. Aside from the simplistic mathematics of a single-assignment iterative step, the algorithm had to check for stopping (by comparing the error of the function given x_n in approximating zero) with a maximum iteration limit check as a backup. The function had to be passed two function objects to represent $f(x)$ and $f'(x)$. Global variables were to be avoided, instead, focusing on a more purely mathematical code base that relied on parameters for pure and consistent behavior. I was explicitly provided the necessary function signature, to be implemented in a file named `newton.cpp`:

```
double newton(Fcn& f, Fcn& df, double x, int maxit, double tol, bool show_iterates)
```

Where `Fcn` was originally an alias for a function pointer, and `show_iterates` was an option to be verbose in the root-finding process.

Testing Testing the Newton's method algorithmic implementation would require testing every combination of the tolerances $\epsilon = \{10^{-1}, 10^{-5}, 10^{-9}\}$ and the first guesses $x_0 = \{-3, 1, 2\}$. This meant nine total tests for a given root finding function $f(x) \equiv x^2(x-3)(x+2) = 0$. All tests were to be run with `show_iterates` enabled for record of the process, which was piped to file.

1.3.4 Implementation

Newton's Method I implemented Newton's method in nine lines of code — the math was extremely simple. Checking conditions like tolerance and maximum iteration added a few lines, but this is not counting the code to display iterate data. One minor change is the redefinition of `Fcn` as an `std::function` object, so as to facilitate the usage of lambdas in this algorithm's parameters.

```
In [5]: %cat ../src/newton.cpp
```

```
//
//  newton.cpp
//  HPSCProject2
//
//  Created by Paul Herz on 9/22/16.
//  Copyright © 2016 Paul Herz. All rights reserved.
//

#include "println.cpp"
```

```
#include <cmath>

using Fcn = std::function<double(double)>;

double newton(
        Fcn& f,
        Fcn& df,
        double x,
        size_t maxit,
        double tol,
        bool show_iterates)
{
        // approximate the root of a provided function to a specified tolerance
        // using Newton's Method.

        // Def. Newton's Method
        //
        //     x_n+1 = x_n - f(x_n) / df(x_n)

        double previous = x;

        for(size_t i = 0; i < maxit; ++i) {

                if(show_iterates) {
                        println("iteration",i,":");
                        println("      x =",x);
                        println("    |h| =", std::abs(previous-x));
                        println(" |f(x)| =", f(x));
                }

                // if within tolerance, end iteration.
                if(tol > std::abs(f(x)/df(x))) {
                        break;
                }

                previous = x;
                x = x - (f(x)/df(x));
        }

        return x;
}
```

Testing Testing was implemented rather quickly as a set of nested loops to iterate all combinations of first guesses and tolerances given. Functions `f` ($f(x)$) and `df` ($f'(x)$) were quickly implemented as C++ Lambdas for readability and portability. An explicit cast to `std::function` was necessary, as the `lambda` type is independent of this.

In [6]: %cat ../src/test_newton.cpp

```
//
//  test_newton.cpp
//  HPSCProject2
//
//  Created by Paul Herz on 9/22/16.
//  Copyright © 2016 Paul Herz. All rights reserved.
//

#include <cmath>

#include "println.cpp"
#include "newton.cpp"

int main(int argc, const char * argv[]) {

        // Test the Newton's Method implementation in `newton.cpp`
        // and determine a root approximation for a given function
        // using its derivative with given tolerance and iteration limit.

        // For your tests, start with initial guesses of x_0 = {−3, 1, 2},
        // and solve the problem to tolerances of ε = {1E−1, 1E−5, 1E−9}
        // (i.e. 9 tests in total). All of these tests should set
        // show iterates to true and should allow a maximum of 50 iterations.

        Fcn f = [](double x) -> double {
                // (x^2)(x^2-x-6)
                double x2 = std::pow(x,2);
                return (x2*(x2-x-6));
        };


        Fcn df = [](double x) -> double {
                // x (4x^2 -3x -12)
                double x2 = std::pow(x,2);
                return (4*x2-3*x-12);
        };

        size_t iterationLimit = 50;

        for(double initialGuess: {-3, 1, 2}) {
                for(double tolerance: {1.e-1, 1.e-5, 1.e-9}) {
                        println("• Initial guess",initialGuess);
                        println("• Tolerance",tolerance);
                        double x = newton(f, df, initialGuess, iterationLimit, tole
                        println("The approximate root is", x);
                        println("\n\n\n");
```

```
                }
        }

        return 0;
}
```

As aforementioned, the output of this routine was piped to a file for demonstrative purposes. It is reproduced below.

Output

```
In [7]: %cat ../data/b_out.txt
```

```
• Initial guess -3
• Tolerance 0.1
iteration 0 :
      x = -3
    |h| = 0
 |f(x)| = 54
iteration 1 :
      x = -4.63636
    |h| = 1.63636
 |f(x)| = 432.76
iteration 2 :
      x = -9.5601
    |h| = 4.92374
 |f(x)| = 8678.52
iteration 3 :
      x = -32.2631
    |h| = 22.703
 |f(x)| = 1.11083e+06
iteration 4 :
      x = -293.732
    |h| = 261.469
 |f(x)| = 7.46878e+09
iteration 5 :
      x = -21880.9
    |h| = 21587.2
 |f(x)| = 2.29234e+17
iteration 6 :
      x = -1.19717e+08
    |h| = 1.19695e+08
 |f(x)| = 2.05408e+32
iteration 7 :
      x = -3.58302e+15
    |h| = 3.58302e+15
 |f(x)| = 1.64814e+62
```

```
iteration 8 :
      x = -3.2095e+30
    |h| = 3.2095e+30
 |f(x)| = 1.06108e+122
iteration 9 :
      x = -2.57522e+60
    |h| = 2.57522e+60
 |f(x)| = 4.39804e+241
iteration 10 :
      x = -1.65794e+120
    |h| = 1.65794e+120
 |f(x)| = inf
iteration 11 :
      x = -inf
    |h| = inf
 |f(x)| = inf
iteration 12 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 13 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 14 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 15 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 16 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 17 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 18 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 19 :
      x = nan
    |h| = nan
 |f(x)| = nan
```

```
iteration 20 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 21 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 22 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 23 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 24 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 25 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 26 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 27 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 28 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 29 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 30 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 31 :
      x = nan
    |h| = nan
 |f(x)| = nan
```

```
iteration 32 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 33 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 34 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 35 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 36 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 37 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 38 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 39 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 40 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 41 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 42 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 43 :
      x = nan
    |h| = nan
 |f(x)| = nan
```

```
iteration 44 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 45 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 46 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 47 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 48 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 49 :
      x = nan
    |h| = nan
 |f(x)| = nan
The approximate root is nan
```

```
• Initial guess -3
• Tolerance 1e-05
iteration 0 :
      x = -3
    |h| = 0
 |f(x)| = 54
iteration 1 :
      x = -4.63636
    |h| = 1.63636
 |f(x)| = 432.76
iteration 2 :
      x = -9.5601
    |h| = 4.92374
 |f(x)| = 8678.52
iteration 3 :
      x = -32.2631
    |h| = 22.703
 |f(x)| = 1.11083e+06
iteration 4 :
```

```
      x = -293.732
    |h| = 261.469
 |f(x)| = 7.46878e+09
iteration 5 :
      x = -21880.9
    |h| = 21587.2
 |f(x)| = 2.29234e+17
iteration 6 :
      x = -1.19717e+08
    |h| = 1.19695e+08
 |f(x)| = 2.05408e+32
iteration 7 :
      x = -3.58302e+15
    |h| = 3.58302e+15
 |f(x)| = 1.64814e+62
iteration 8 :
      x = -3.2095e+30
    |h| = 3.2095e+30
 |f(x)| = 1.06108e+122
iteration 9 :
      x = -2.57522e+60
    |h| = 2.57522e+60
 |f(x)| = 4.39804e+241
iteration 10 :
      x = -1.65794e+120
    |h| = 1.65794e+120
 |f(x)| = inf
iteration 11 :
      x = -inf
    |h| = inf
 |f(x)| = inf
iteration 12 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 13 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 14 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 15 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 16 :
```

```
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 17 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 18 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 19 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 20 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 21 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 22 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 23 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 24 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 25 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 26 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 27 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 28 :
```

```
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 29 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 30 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 31 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 32 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 33 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 34 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 35 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 36 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 37 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 38 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 39 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 40 :
```

```
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 41 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 42 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 43 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 44 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 45 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 46 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 47 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 48 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 49 :
      x = nan
    |h| = nan
 |f(x)| = nan
The approximate root is nan
```

```
• Initial guess -3
• Tolerance 1e-09
iteration 0 :
      x = -3
```

```
    |h| = 0
 |f(x)| = 54
iteration 1 :
      x = -4.63636
    |h| = 1.63636
 |f(x)| = 432.76
iteration 2 :
      x = -9.5601
    |h| = 4.92374
 |f(x)| = 8678.52
iteration 3 :
      x = -32.2631
    |h| = 22.703
 |f(x)| = 1.11083e+06
iteration 4 :
      x = -293.732
    |h| = 261.469
 |f(x)| = 7.46878e+09
iteration 5 :
      x = -21880.9
    |h| = 21587.2
 |f(x)| = 2.29234e+17
iteration 6 :
      x = -1.19717e+08
    |h| = 1.19695e+08
 |f(x)| = 2.05408e+32
iteration 7 :
      x = -3.58302e+15
    |h| = 3.58302e+15
 |f(x)| = 1.64814e+62
iteration 8 :
      x = -3.2095e+30
    |h| = 3.2095e+30
 |f(x)| = 1.06108e+122
iteration 9 :
      x = -2.57522e+60
    |h| = 2.57522e+60
 |f(x)| = 4.39804e+241
iteration 10 :
      x = -1.65794e+120
    |h| = 1.65794e+120
 |f(x)| = inf
iteration 11 :
      x = -inf
    |h| = inf
 |f(x)| = inf
iteration 12 :
      x = nan
```

```
    |h| = nan
 |f(x)| = nan
iteration 13 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 14 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 15 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 16 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 17 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 18 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 19 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 20 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 21 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 22 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 23 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 24 :
      x = nan
```

```
    |h| = nan
 |f(x)| = nan
iteration 25 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 26 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 27 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 28 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 29 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 30 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 31 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 32 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 33 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 34 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 35 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 36 :
      x = nan
```

```
    |h| = nan
 |f(x)| = nan
iteration 37 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 38 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 39 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 40 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 41 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 42 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 43 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 44 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 45 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 46 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 47 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 48 :
      x = nan
```

```
    |h| = nan
 |f(x)| = nan
iteration 49 :
      x = nan
    |h| = nan
 |f(x)| = nan
The approximate root is nan
```

```
• Initial guess 1
• Tolerance 0.1
iteration 0 :
      x = 1
    |h| = 0
 |f(x)| = -6
iteration 1 :
      x = 0.454545
    |h| = 0.545455
 |f(x)| = -1.2909
iteration 2 :
      x = 0.35158
    |h| = 0.102965
 |f(x)| = -0.769831
The approximate root is 0.35158
```

```
• Initial guess 1
• Tolerance 1e-05
iteration 0 :
      x = 1
    |h| = 0
 |f(x)| = -6
iteration 1 :
      x = 0.454545
    |h| = 0.545455
 |f(x)| = -1.2909
iteration 2 :
      x = 0.35158
    |h| = 0.102965
 |f(x)| = -0.769831
iteration 3 :
      x = 0.290289
    |h| = 0.0612908
 |f(x)| = -0.522969
```

```
iteration 4 :
      x = 0.248565
    |h| = 0.0417247
 |f(x)| = -0.382247
iteration 5 :
      x = 0.217981
    |h| = 0.0305833
 |f(x)| = -0.293195
iteration 6 :
      x = 0.194458
    |h| = 0.0235236
 |f(x)| = -0.232806
iteration 7 :
      x = 0.175732
    |h| = 0.0187262
 |f(x)| = -0.189763
iteration 8 :
      x = 0.160433
    |h| = 0.0152989
 |f(x)| = -0.157899
iteration 9 :
      x = 0.147677
    |h| = 0.0127561
 |f(x)| = -0.133595
iteration 10 :
      x = 0.136864
    |h| = 0.0108124
 |f(x)| = -0.114604
iteration 11 :
      x = 0.127574
    |h| = 0.00929045
 |f(x)| = -0.0994619
iteration 12 :
      x = 0.119499
    |h| = 0.00807477
 |f(x)| = -0.0871827
iteration 13 :
      x = 0.112412
    |h| = 0.00708723
 |f(x)| = -0.0770794
iteration 14 :
      x = 0.106138
    |h| = 0.0062734
 |f(x)| = -0.068661
iteration 15 :
      x = 0.100544
    |h| = 0.00559431
 |f(x)| = -0.061569
```

```
iteration 16 :
      x = 0.0955227
    |h| = 0.00502145
 |f(x)| = -0.0555358
iteration 17 :
      x = 0.0909892
    |h| = 0.00453351
 |f(x)| = -0.0503589
iteration 18 :
      x = 0.0868748
    |h| = 0.00411434
 |f(x)| = -0.0458821
iteration 19 :
      x = 0.0831234
    |h| = 0.00375147
 |f(x)| = -0.0419836
iteration 20 :
      x = 0.0796882
    |h| = 0.00343516
 |f(x)| = -0.038567
iteration 21 :
      x = 0.0765305
    |h| = 0.00315769
 |f(x)| = -0.0355554
iteration 22 :
      x = 0.0736176
    |h| = 0.00291291
 |f(x)| = -0.0328869
iteration 23 :
      x = 0.0709218
    |h| = 0.00269583
 |f(x)| = -0.0305108
iteration 24 :
      x = 0.0684194
    |h| = 0.0025024
 |f(x)| = -0.0283856
iteration 25 :
      x = 0.0660901
    |h| = 0.00232926
 |f(x)| = -0.026477
iteration 26 :
      x = 0.0639164
    |h| = 0.00217367
 |f(x)| = -0.0247563
iteration 27 :
      x = 0.0618831
    |h| = 0.0020333
 |f(x)| = -0.0231995
```

```
iteration 28 :
      x = 0.0599769
    |h| = 0.00190623
 |f(x)| = -0.0217862
iteration 29 :
      x = 0.0581861
    |h| = 0.00179081
 |f(x)| = -0.0204993
iteration 30 :
      x = 0.0565004
    |h| = 0.00168565
 |f(x)| = -0.019324
iteration 31 :
      x = 0.0549109
    |h| = 0.00158957
 |f(x)| = -0.0182477
iteration 32 :
      x = 0.0534093
    |h| = 0.00150154
 |f(x)| = -0.0172596
iteration 33 :
      x = 0.0519887
    |h| = 0.00142068
 |f(x)| = -0.0163501
iteration 34 :
      x = 0.0506424
    |h| = 0.00134623
 |f(x)| = -0.0155112
iteration 35 :
      x = 0.0493649
    |h| = 0.00127752
 |f(x)| = -0.0147357
iteration 36 :
      x = 0.0481509
    |h| = 0.00121398
 |f(x)| = -0.0140173
iteration 37 :
      x = 0.0469958
    |h| = 0.0011551
 |f(x)| = -0.0133506
iteration 38 :
      x = 0.0458954

 |f(x)| = -0.0127306
iteration 39 :
      x = 0.0448458
    |h| = 0.00104957
 |f(x)| = -0.012153
```

```
iteration 40 :
      x = 0.0438436
    |h| = 0.00100219
 |f(x)| = -0.0116142
iteration 41 :
      x = 0.0428857
    |h| = 0.000957961
 |f(x)| = -0.0111106
iteration 42 :
      x = 0.0419691
    |h| = 0.000916616
 |f(x)| = -0.0106392
iteration 43 :
      x = 0.0410912
    |h| = 0.000877907
 |f(x)| = -0.0101974
iteration 44 :
      x = 0.0402495
    |h| = 0.000841614
 |f(x)| = -0.00978273
iteration 45 :
      x = 0.039442
    |h| = 0.000807538
 |f(x)| = -0.00939297
iteration 46 :
      x = 0.0386665
    |h| = 0.000775503
 |f(x)| = -0.00902616
iteration 47 :
      x = 0.0379212
    |h| = 0.000745347
 |f(x)| = -0.00868055
iteration 48 :
      x = 0.0372042
    |h| = 0.000716926
 |f(x)| = -0.00835451
iteration 49 :
      x = 0.0365141
    |h| = 0.000690109
 |f(x)| = -0.00804659
The approximate root is 0.0358493
```

- Initial guess 1
- Tolerance 1e-09

```
iteration 0 :
```

```
      x = 1
    |h| = 0
 |f(x)| = -6
iteration 1 :
      x = 0.454545
    |h| = 0.545455
 |f(x)| = -1.2909
iteration 2 :
      x = 0.35158
    |h| = 0.102965
 |f(x)| = -0.769831
iteration 3 :
      x = 0.290289
    |h| = 0.0612908
 |f(x)| = -0.522969
iteration 4 :
      x = 0.248565
    |h| = 0.0417247
 |f(x)| = -0.382247
iteration 5 :
      x = 0.217981
    |h| = 0.0305833
 |f(x)| = -0.293195
iteration 6 :
      x = 0.194458
    |h| = 0.0235236
 |f(x)| = -0.232806
iteration 7 :
      x = 0.175732
    |h| = 0.0187262
 |f(x)| = -0.189763
iteration 8 :
      x = 0.160433
    |h| = 0.0152989
 |f(x)| = -0.157899
iteration 9 :
      x = 0.147677
    |h| = 0.0127561
 |f(x)| = -0.133595
iteration 10 :
      x = 0.136864
    |h| = 0.0108124
 |f(x)| = -0.114604
iteration 11 :
      x = 0.127574
    |h| = 0.00929045
 |f(x)| = -0.0994619
iteration 12 :
```

```
      x = 0.119499
    |h| = 0.00807477
 |f(x)| = -0.0871827
iteration 13 :
      x = 0.112412
    |h| = 0.00708723
 |f(x)| = -0.0770794
iteration 14 :
      x = 0.106138
    |h| = 0.0062734
 |f(x)| = -0.068661
iteration 15 :
      x = 0.100544
    |h| = 0.00559431
 |f(x)| = -0.061569
iteration 16 :
      x = 0.0955227
    |h| = 0.00502145
 |f(x)| = -0.0555358
iteration 17 :
      x = 0.0909892
    |h| = 0.00453351
 |f(x)| = -0.0503589
iteration 18 :
      x = 0.0868748
    |h| = 0.00411434
 |f(x)| = -0.0458821
iteration 19 :
      x = 0.0831234
    |h| = 0.00375147
 |f(x)| = -0.0419836
iteration 20 :
      x = 0.0796882
    |h| = 0.00343516
 |f(x)| = -0.038567
iteration 21 :
      x = 0.0765305
    |h| = 0.00315769
 |f(x)| = -0.0355554
iteration 22 :
      x = 0.0736176
    |h| = 0.00291291
 |f(x)| = -0.0328869
iteration 23 :
      x = 0.0709218
    |h| = 0.00269583
 |f(x)| = -0.0305108
iteration 24 :
```

```
      x = 0.0684194
    |h| = 0.0025024
 |f(x)| = -0.0283856
iteration 25 :
      x = 0.0660901
    |h| = 0.00232926
 |f(x)| = -0.026477
iteration 26 :
      x = 0.0639164
    |h| = 0.00217367
 |f(x)| = -0.0247563
iteration 27 :
      x = 0.0618831
    |h| = 0.0020333
 |f(x)| = -0.0231995
iteration 28 :
      x = 0.0599769
    |h| = 0.00190623
 |f(x)| = -0.0217862
iteration 29 :
      x = 0.0581861
    |h| = 0.00179081
 |f(x)| = -0.0204993
iteration 30 :
      x = 0.0565004
    |h| = 0.00168565
 |f(x)| = -0.019324
iteration 31 :
      x = 0.0549109
    |h| = 0.00158957
 |f(x)| = -0.0182477
iteration 32 :
      x = 0.0534093
    |h| = 0.00150154
 |f(x)| = -0.0172596
iteration 33 :
      x = 0.0519887
    |h| = 0.00142068
 |f(x)| = -0.0163501
iteration 34 :
      x = 0.0506424
    |h| = 0.00134623
 |f(x)| = -0.0155112
iteration 35 :
      x = 0.0493649
    |h| = 0.00127752
 |f(x)| = -0.0147357
iteration 36 :
```

```
      x = 0.0481509
    |h| = 0.00121398
 |f(x)| = -0.0140173
iteration 37 :
      x = 0.0469958
    |h| = 0.0011551
 |f(x)| = -0.0133506
iteration 38 :
      x = 0.0458954
    |h| = 0.00110043
 |f(x)| = -0.0127306
iteration 39 :
      x = 0.0448458
    |h| = 0.00104957
 |f(x)| = -0.012153
iteration 40 :
      x = 0.0438436
    |h| = 0.00100219
 |f(x)| = -0.0116142
iteration 41 :
      x = 0.0428857
    |h| = 0.000957961
 |f(x)| = -0.0111106
iteration 42 :
      x = 0.0419691
    |h| = 0.000916616
 |f(x)| = -0.0106392
iteration 43 :
      x = 0.0410912
    |h| = 0.000877907
 |f(x)| = -0.0101974
iteration 44 :
      x = 0.0402495
    |h| = 0.000841614
 |f(x)| = -0.00978273
iteration 45 :
      x = 0.039442
    |h| = 0.000807538
 |f(x)| = -0.00939297
iteration 46 :
      x = 0.0386665
    |h| = 0.000775503
 |f(x)| = -0.00902616
iteration 47 :
      x = 0.0379212
    |h| = 0.000745347
 |f(x)| = -0.00868055
iteration 48 :
```

```
      x = 0.0372042
    |h| = 0.000716926
 |f(x)| = -0.00835451
iteration 49 :
      x = 0.0365141
    |h| = 0.000690109
 |f(x)| = -0.00804659
The approximate root is 0.0358493
```

- Initial guess 2
- Tolerance 0.1

```
iteration 0 :
      x = 2
    |h| = 0
 |f(x)| = -16
iteration 1 :
      x = -6
    |h| = 8
 |f(x)| = 1296
iteration 2 :
      x = -14.64
    |h| = 8.64
 |f(x)| = 47789
iteration 3 :
      x = -68.3815
    |h| = 53.7415
 |f(x)| = 2.21569e+07
iteration 4 :
      x = -1240.88
    |h| = 1172.49
 |f(x)| = 2.3728e+12
iteration 5 :
      x = -386261
    |h| = 385020
 |f(x)| = 2.22599e+22
iteration 6 :
      x = -3.72998e+10
    |h| = 3.72994e+10
 |f(x)| = 1.93564e+42
iteration 7 :
      x = -3.47818e+20
    |h| = 3.47818e+20
 |f(x)| = 1.46356e+82
iteration 8 :
      x = -3.02444e+40
```

```
    |h| = 3.02444e+40
 |f(x)| = 8.3672e+161
iteration 9 :
      x = -2.28681e+80
    |h| = 2.28681e+80
 |f(x)| = inf
iteration 10 :
      x = -inf
    |h| = inf
 |f(x)| = inf
iteration 11 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 12 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 13 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 14 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 15 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 16 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 17 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 18 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 19 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 20 :
      x = nan
```

```
    |h| = nan
 |f(x)| = nan
iteration 21 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 22 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 23 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 24 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 25 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 26 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 27 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 28 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 29 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 30 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 31 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 32 :
      x = nan
```

```
    |h| = nan
 |f(x)| = nan
iteration 33 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 34 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 35 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 36 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 37 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 38 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 39 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 40 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 41 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 42 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 43 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 44 :
      x = nan
```

```
    |h| = nan
 |f(x)| = nan
iteration 45 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 46 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 47 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 48 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 49 :
      x = nan
    |h| = nan
 |f(x)| = nan
The approximate root is nan
```

```
• Initial guess 2
• Tolerance 1e-05
iteration 0 :
      x = 2
    |h| = 0
 |f(x)| = -16
iteration 1 :
      x = -6
    |h| = 8
 |f(x)| = 1296
iteration 2 :
      x = -14.64
    |h| = 8.64
 |f(x)| = 47789
iteration 3 :
      x = -68.3815
    |h| = 53.7415
 |f(x)| = 2.21569e+07
iteration 4 :
      x = -1240.88
    |h| = 1172.49
```

```
 |f(x)| = 2.3728e+12
iteration 5 :
      x = -386261
    |h| = 385020
 |f(x)| = 2.22599e+22
iteration 6 :
      x = -3.72998e+10
    |h| = 3.72994e+10
 |f(x)| = 1.93564e+42
iteration 7 :
      x = -3.47818e+20
    |h| = 3.47818e+20
 |f(x)| = 1.46356e+82
iteration 8 :
      x = -3.02444e+40
    |h| = 3.02444e+40
 |f(x)| = 8.3672e+161
iteration 9 :
      x = -2.28681e+80
    |h| = 2.28681e+80
 |f(x)| = inf
iteration 10 :
      x = -inf
    |h| = inf
 |f(x)| = inf
iteration 11 :
      x = nan
    |h| = nan
 |f(x)| = nan

      x = nan
    |h| = nan
 |f(x)| = nan
iteration 13 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 14 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 15 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 16 :
      x = nan
    |h| = nan
```

```
 |f(x)| = nan
iteration 17 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 18 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 19 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 20 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 21 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 22 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 23 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 24 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 25 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 26 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 27 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 28 :
      x = nan
    |h| = nan
```

```
 |f(x)| = nan
iteration 29 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 30 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 31 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 32 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 33 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 34 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 35 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 36 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 37 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 38 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 39 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 40 :
      x = nan
    |h| = nan
```

```
 |f(x)| = nan
iteration 41 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 42 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 43 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 44 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 45 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 46 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 47 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 48 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 49 :
      x = nan
    |h| = nan
 |f(x)| = nan
The approximate root is nan
```

```
• Initial guess 2
• Tolerance 1e-09
iteration 0 :
      x = 2
    |h| = 0
 |f(x)| = -16
```

```
iteration 1 :
      x = -6
    |h| = 8
 |f(x)| = 1296
iteration 2 :
      x = -14.64
    |h| = 8.64
 |f(x)| = 47789
iteration 3 :
      x = -68.3815
    |h| = 53.7415
 |f(x)| = 2.21569e+07
iteration 4 :
      x = -1240.88
    |h| = 1172.49
 |f(x)| = 2.3728e+12
iteration 5 :
      x = -386261
    |h| = 385020
 |f(x)| = 2.22599e+22
iteration 6 :
      x = -3.72998e+10
    |h| = 3.72994e+10
 |f(x)| = 1.93564e+42
iteration 7 :
      x = -3.47818e+20
    |h| = 3.47818e+20
 |f(x)| = 1.46356e+82
iteration 8 :
      x = -3.02444e+40
    |h| = 3.02444e+40
 |f(x)| = 8.3672e+161
iteration 9 :
      x = -2.28681e+80
    |h| = 2.28681e+80
 |f(x)| = inf
iteration 10 :
      x = -inf
    |h| = inf
 |f(x)| = inf
iteration 11 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 12 :
      x = nan
    |h| = nan
 |f(x)| = nan
```

```
iteration 13 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 14 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 15 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 16 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 17 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 18 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 19 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 20 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 21 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 22 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 23 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 24 :
      x = nan
    |h| = nan
 |f(x)| = nan
```

```
iteration 25 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 26 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 27 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 28 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 29 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 30 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 31 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 32 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 33 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 34 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 35 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 36 :
      x = nan
    |h| = nan
 |f(x)| = nan
```

```
iteration 37 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 38 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 39 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 40 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 41 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 42 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 43 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 44 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 45 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 46 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 47 :
      x = nan
    |h| = nan
 |f(x)| = nan
iteration 48 :
      x = nan
    |h| = nan
 |f(x)| = nan
```

```
iteration 49 :
      x = nan
    |h| = nan
 |f(x)| = nan
The approximate root is nan
```

1.3.5 Analysis

Note that only the tests where the initial guess was 1 result in a final value — the other guesses, regardless of tolerance, diverge off into the realm of computationally incomprehensible numbers quickly, supplanted with `NaN` (not a number) or `inf` (infinity) rather quickly, instead of closing in on (converging upon) a number. It is interesting how such a small difference between initial guesses is enough to be considered an acceptably close guess vs. a poor guess relative to the performance of Newton's method, which depends rather directly on these guesses. For all tests where the guess was 1 (one), a converging guess, adjusting the tolerance caused the test to require more iterations, all but the first, biggest tolerance requiring maximum iterations before being cut off. They also led to a higher degree of accuracy up until the termination point.

1.4 Part C – Kepler's Equation and Orbits

Part C of this project applies the mathematics and lessons from the prior section in a real-world astronomy scenario that consists of simulating an orbit relative to time given measures that produce orbital eccentricity, a time-proportional constant, and formulae relating to orbital position and mapping conversion.

1.4.1 Goals

Over a period of time, represented by the linear span $\{0, 0.001, \ldots, 10\}$ for the time-proportional variable t, I was to calculate the orbital angle of the planetary body in question, given data for its eccentricity. I then had to map this polar data to a Cartesian system so as to graph it and prove the accuracy of my calculation.

1.4.2 Math Background

Orbital Eccentricity Orbital eccentricity is a measure of the focal deviation of an ellipse from a single-focus circle, given measures a and b.

$$\epsilon = \sqrt{1 - \frac{b^2}{a^2}}$$

Kepler's Equation Given orbital eccentricity ϵ, and one of: (1) ω, the elliptical orbital angle, or (2) t, a time-proportional constant, calculate the other, i.e. the unknown:

$$\epsilon \sin(\omega) - \omega = t$$

True Anomaly to Body Radius True anomaly is one of many orbital elements: pieces of information required for fully describing a Keplerian orbit in three dimensional space. The true anomaly (normally written ν, here written ω, not to be confused with the argument of the periapsis) represents the angle relative to a reference angle 0 of the orbiting body on the elliptical path. It is important to be able to calculate the radius, i.e. the distance of the orbiting body from the orbital center, as a function of the true anomaly.

$$r(\omega) = \frac{ab}{\sqrt{(b\cos(\omega))^2 + (a\sin(\omega))^2}}$$

Cartesian Conversion of Polar Coordinates To graph the results of the elliptical orbit function, we must perform a map of the polar coordinate system to the Cartesian system that we use here. This is done simply, given a polar coordinate (r, ω) representing radius and angle:

$$x = r\cos(\omega),\ y = r\sin(\omega)$$

1.4.3 Implementation

Kepler's Equation Given Kepler's equation, relating a time constant to orbital angle, I produced a routine to iterate over a linear span for the time constant and approximate ω to a sufficient level of precision using Newton's method, and using the previous value of ω as a first guess for the next value.

I separated concerns like eccentricity calculation and control constants like Newton tolerance and maximum iterations to keep the code short and readable. Kepler's equation was adapted to the form $0 = \epsilon\sin(\omega) - \omega - t$ so as to be in proper form for Newton's algorithm. The code is mostly calculational, save for the use of `Vector::mapElements` to iterate over the linear span for t. All other code is either self-explanatory due to its mathematical nature, or well documented. Note that in this routine, unlike the prior two, data was persisted to file. This was for graphing purposes in this document.

Newton's Method Please refer above to the dedicated section on Newton's method for detail on my implementation.

```
In [8]: %cat ../src/kepler.cpp
```

```
//
//  kepler.cpp
//  HPSCProject2
//
//  Created by Paul Herz on 10/3/16.
//  Copyright © 2016 Paul Herz. All rights reserved.
//

#include <cmath>
#include <vector>
#include "println.cpp"
#include "Vector.h"
#include "newton.cpp"
```

```
using namespace PH;

// Kepler's equation:
//
//     ε sin(ω) - ω = t
//
// Where:
// • ε = sqrt(1 - b^2 / a^2) [orbital eccentricity]
// • t is proportional to time
// • ω is the angle of the object about its elliptical orbit

double eccentricity(double a, double b) {
        return std::sqrt( 1 - std::pow(b,2) / std::pow(a,2) );
}

double r(double ω, double a, double b) {
        return ( (a*b) / sqrt( pow(b*cos(ω),2) + pow(a*sin(ω),2)) );
}

// nonlinear root-finding function to solve
// Kepler's for ω
double f(double ω, double t, double ε) {
        //  0 = ε sin(ω) - ω - t
        return ((ε * sin(ω)) - ω - t);
}

double f1(double ω, double ε) {
        //  0 = ε cos(ω) - 1
        return (ε * cos(ω) - 1.0);
}

int main(int argc, const char * argv[]) {

        // Given values
        double a = 2.0;
        double b = 1.25;
        double ε = eccentricity(a, b);
        double tolerance = 1.e-5;
        size_t maxIterations = 6;
        size_t timeSpaceSize = 10000;

        auto newtonResults = Vector(timeSpaceSize);
        auto x_t = Vector(timeSpaceSize);
        auto y_t = Vector(timeSpaceSize);

        // For each time t = {0, 0.001, ... , 10}:
        auto tSpace = Vector::linSpace(0, 10, timeSpaceSize);
```

```
        double guess = 0;
        tSpace.mapElements([&](double t, double index){

                // • use Newton's method to solve Kepler's equation
                //   for ω(t). Tolerance 1e-5, max 6 iterations,
                //   no output. Guess should be previous solution or
                //   initially zero. (no global variables)

                // Unary versions of f() and f'() for compatibility
                // with newton(), which expects (double)->double.

                Fcn  _f = [&](double ω){ return f(ω,t,ε); };
                Fcn _f1 = [&](double ω){ return f1(ω,ε); };

                double ω = newton(_f, _f1, guess, maxIterations, tolerance, false);
                guess = ω;
                newtonResults[index] = ω;

                // • use the formula for radial position of an object
                //   at angle ω to compute cartesian coordinates
                //   x(t) = r(ω) cos(ω) and y(t) = r(ω) sin(ω).

                double r_ω = r(ω, a, b);

                x_t[index] = r_ω * cos(ω);
                y_t[index] = r_ω * sin(ω);

        });

        // • store the sets t, x(t), y(t), then export them.

        std::string prefix = "../data/";
        tSpace.saveTo(prefix + "t.txt");
        x_t.saveTo(prefix + "x.txt");
        y_t.saveTo(prefix + "y.txt");

}
```

1.4.4 Analysis

Newton's algorithm did a remarkably good job at achieving high-resolution datapoints, as is apparent in the graphs below — anecdote and cursory analysis seem to demonstrate that the data generated is an accurate elliptical orbit over time. I invite you to refer yourself or a friend astrophysicist to the graphs that follow.

```
In [9]: %pylab inline
        pylab.rcParams['figure.figsize'] = (10, 6)
```

```
        matplotlib.rcParams.update({'font.size': 16})
        matplotlib.rcParams.update({'axes.labelsize': 20})
        matplotlib.rcParams.update({'xtick.labelsize': 12})
        matplotlib.rcParams.update({'ytick.labelsize': 12})
        matplotlib.rcParams.update({
                'font.family': 'Helvetica, Arial, sans-serif'
        })

        %config InlineBackend.figure_format = 'retina'
```

```
Populating the interactive namespace from numpy and matplotlib
```

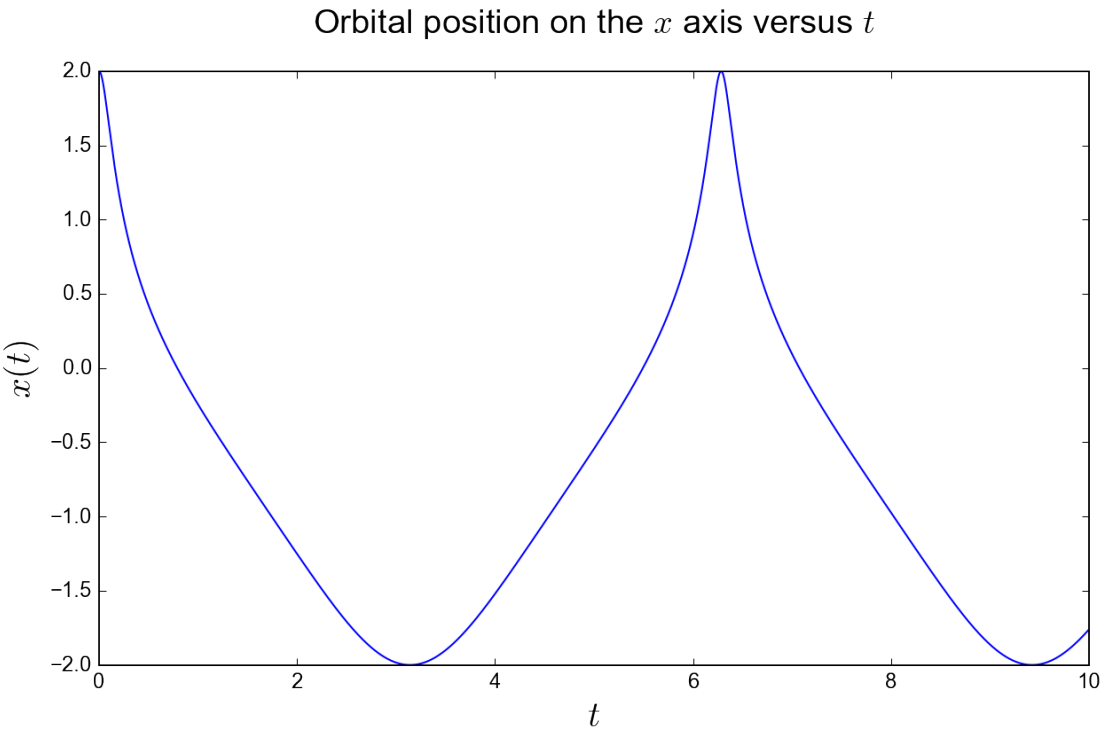
In [10]:
```
names = ['t','x','y']
v = {}
for name in names:
    v[name] = loadtxt('../data/' + name + '.txt')
```

In [11]:
```
# Create three plots:
# 1. x(t) vs t

pylab.plot(v['t'], v['x'])

pylab.xlabel('$t$')
pylab.ylabel('$x(t)$')
pylab.title('Orbital position on the $x$ axis versus $t$', y=1.05)
```

Out[11]:
```
<matplotlib.text.Text at 0x1045057b8>
```

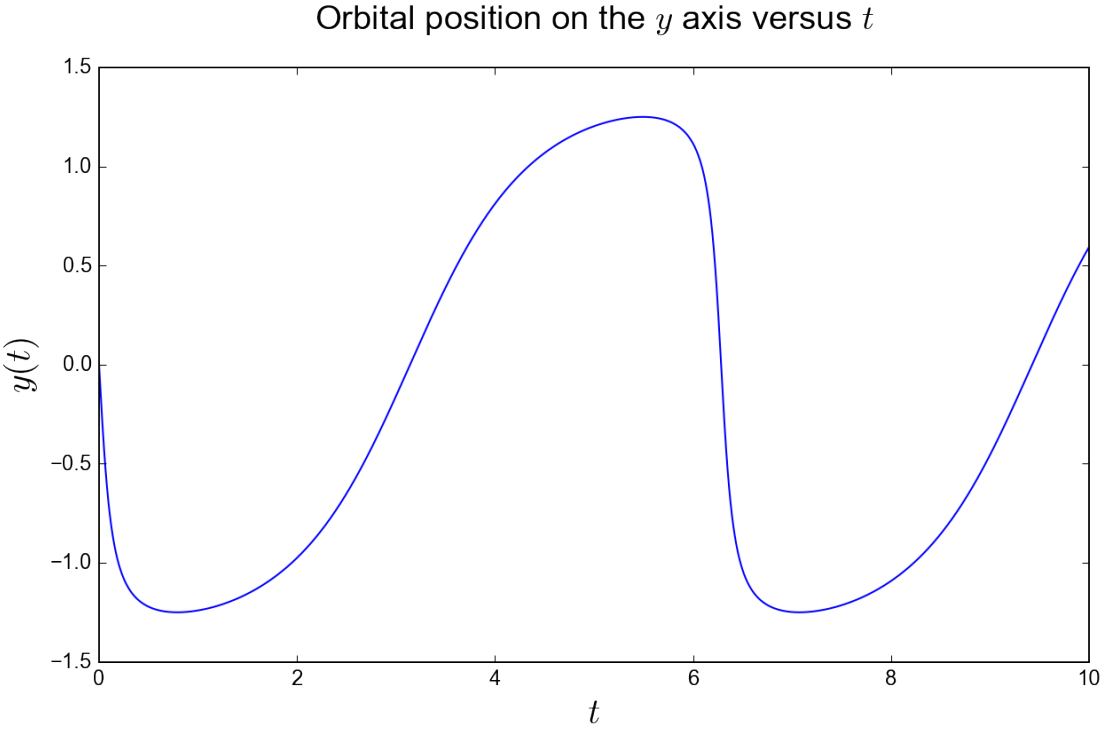


In [12]:
```
# 2. y(t) vs t

pylab.plot(v['t'], v['y'])

pylab.xlabel('$t$')
pylab.ylabel('$y(t)$')
pylab.title('Orbital position on the $y$ axis versus $t$', y=1.05)
```

Out[12]: <matplotlib.text.Text at 0x10444d198>

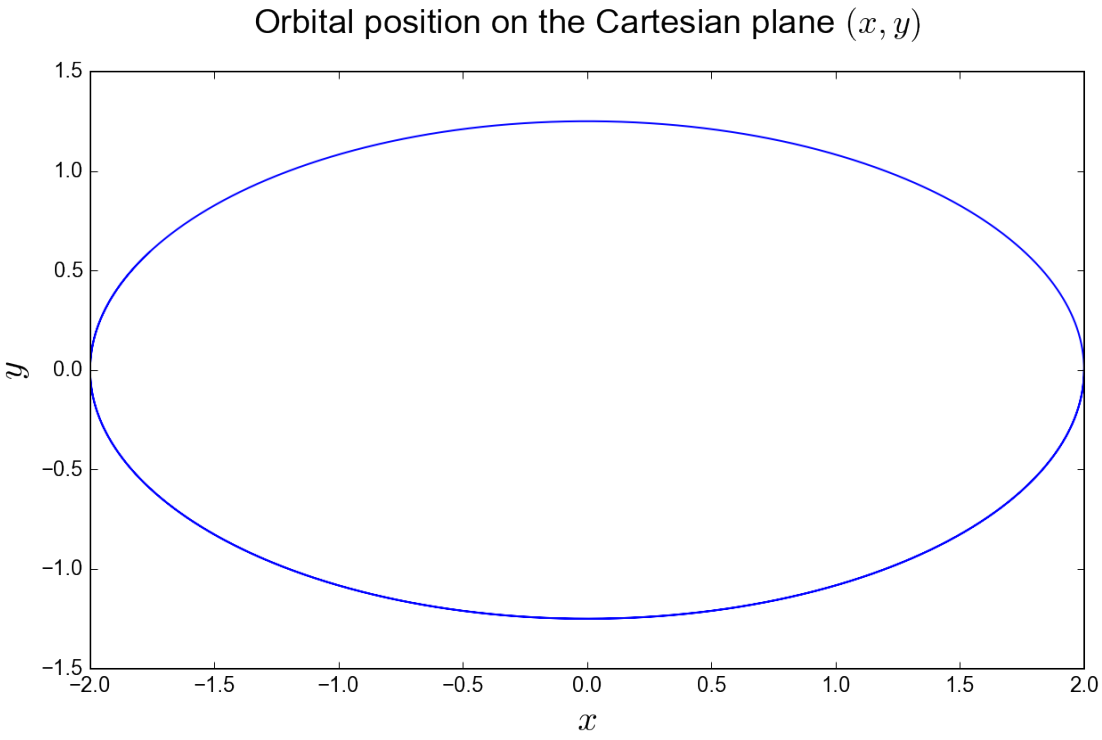


In [13]:
```
# 3. y(t) vs x(t)

pylab.plot(v['x'], v['y'])

pylab.xlabel('$x$')
pylab.ylabel('$y$')
pylab.title('Orbital position on the Cartesian plane $(x,y)$', y=1.05)
```

Out[13]: <matplotlib.text.Text at 0x104451ac8>


Orbital position on the Cartesian plane (x, y)
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
y
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
x